BARTŁOMIEJ FILIPEK

C++17 IN DETAIL

PART I - LANGUAGE FEATURES

LEARN THE EXCITING FEATURES OF
THE NEW C++ STANDARD!

(BF)
C++ STORIES

BFILIPEK.COM

C++17 in Detail

Learn the Exciting Features of The New C++ Standard!

Bartłomiej Filipek

This book is for sale at http://leanpub.com/cpp17indetail

This version was published on 2019-11-13

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

for Wiola and Mikołaj

Contents

About the Author

Bartłomiej (Bartek) Filipek is a C++ software developer with more than 12 years of professional experience. In 2010 he graduated from Jagiellonian University in Cracow, Poland with a Masters Degree in Computer Science.

Bartek currently works at Xara, where he develops features for advanced document editors. He also has experience with desktop graphics applications, game development, large-scale systems for aviation, writing graphics drivers and even biofeedback. In the past, Bartek has also taught programming (mostly game and graphics programming courses) at local universities in Cracow.

Since 2011 Bartek has been regularly blogging at bfilipek.com. Initially, the topics revolved around graphics programming, but now the blog focuses on core C++. He's also a co-organiser of the C++ User Group in Cracow. You can hear Bartek in one @CppCast episode where he talks about C++17, blogging and text processing.

Since October 2018, Bartek has been a C++ Expert for the Polish National Body which works directly with ISO/IEC JTC 1/SC 22 (C++ Standardisation Committee). In the same month, Bartek was awarded his first MVP title for the years 2019/2020 by Microsoft.

In his spare time, he loves collecting and assembling Lego models with his little son.

Technical Reviewer

Jacek Galowicz is a Software Engineer with roughly a decade of professional experience in C++. He got his master of science degree in electrical engineering at RWTH Aachen University in Germany.

Jacek co-founded the Cyberus Technology GmbH in early 2017 and works on products around low-level cybersecurity, virtualization, microkernels, and advanced testing infrastructure. At former jobs, he implemented performance- and security-sensitive microkernel operating systems for Intel x86 virtualization at Intel and FireEye in Germany. In general, he gained experience with kernel driver development, 3D graphics programming, databases, network communication, physics simulation, mostly in C or C++.

In his free time, Jacek maintains a little C++ blog, which has seen some lack of love while he wrote the C++17 STL Cookbook. He is a regular visitor of the C++ Usergroups in Hannover and Braunschweig. In order to do meta programming and generic programming better, he also learned and uses Haskell, which in turn sparked his interest to generally bring the advantages of purely functional programming to C++.

Additional Reviewers & Supporters

Without the support of many good people, this book would have been far less than it is. It is a great pleasure to thank them. A lot of people read drafts, found errors, pointed out confusing explanations, suggested different wording, tested programs, and offered support and encouragement. Many reviewers generously supplied insights and comments that I was able to incorporate into the book. Any mistakes that remain are, of course, my own.

Thanks especially to the following reviewers, who either commented on large sections of the book, smaller parts or gave me a general direction for the whole project.

Patrice Roy - Patrice has been playing with C++, either professionally, for pleasure or (most of the time) both for over 20 years. After a few years doing R&D and working on military flight simulators, he moved on to academics and has been teaching computer science since 1998. Since 2005, he's been involved more specifically in helping graduate students and professionals from the fields of real-time systems and game programming develop the skills they need to face today's challenges. The rapid evolution of C++ in recent years has made his job even more enjoyable.

Jonathan Boccara - Jonathan is a passionate C++ developer working on large codebase of financial software. His interests revolve around making code expressive. He created and regularly blogs on Fluent C++[1], where he explores how to use the C++ language to write expressive code, make existing code clearer, and also about how to keep your spirits up when facing unclear code.

Łukasz Rachwalski - Software engineer - founder C++ User Group Krakow.

Michał Czaja - C++ software developer and network engineer. Works in telecommunication industry since 2009.

Arne Mertz - Software Engineer from Hamburg, Germany. He is a C++ and clean code enthusiast. He's the author of the Simplify C++[2] blog.

JFT - Has been involved with computer programming and "computer techy stuff" for over 45 years - including OS development and teaching c++ in the mid 1990's.

[1] https://www.fluentcpp.com/
[2] https://arne-mertz.de/

Victor Ciura - Senior Software Engineer at CAPHYON and Technical Lead on the Advanced Installer team[3]. For over a decade, he designed and implemented several core components and libraries of Advanced Installer. He's a regular guest at Computer Science Department of his Alma Mater, University of Craiova, where he gives student lectures & workshops on "Using C++STL for Competitive Programming and Software Development". Currently, he spends most of his time working with his team on improving and extending the repackaging and virtualization technologies in Advanced Installer IDE, helping clients migrate their Win32 desktop apps to the Windows Store (MSIX).

Karol Gasiński - Tech Lead on macOS VR in Apple's GPU SW Architecture Team. Previously Senior Graphics Software Engineer at Intel. As a member of KHRONOS group, he contributed to OpenGL 4.4 and OpenGL ES 3.1 Specifications. Except for his work, Karol's biggest passion is game development and low-level programming. He conducts research in the field of VR, new interfaces and game engines. In game industry is known from founding WGK conference. In the past, he was working on mobile versions of such games as Medieval Total War, Pro Evolution Soccer and Silent Hill.

Marco Arena - Software Engineer and C++ Specialist building mission critical and high performance software products in the Formula 1 Industry. Marco is very active in the C++ ecosystem as a blogger, speaker and organizer: he founded the Italian C++ Community in 2013, he joined isocpp.org[4] editors in 2014 and he has been involved in community activities for many years. Marco has held the Microsoft MVP Award since 2016. Discover more at marcoarena.com[5].

Konrad Jaśkowiec - C++ expert with almost 8 years of professional experience at the time with prime focus on system design and optimization. You can find his profile at Linkedin[6].

[3] http://www.advancedinstaller.com
[4] http://isocpp.org/
[5] http://marcoarena.com/
[6] https://www.linkedin.com/in/konrad-ja%C5%9Bkowiec-84585159/

Daniel Khoshnoudirad - a passionate C++ Developer. Daniel graduated in 2016 with the PhD in Computer Science from Université Paris-Est, under the direction of Pr. Hugues Talbot. Daniel also holds a Master's degree in Mathematical Engineering from Université de Bordeaux. He is a proud reviewer of the French version of Effective Modern C++ by Dr Scott Meyers. He has experience in Qt, Python, Fortran, Machine Learning and teaching. Daniel is also interested in Java, Rust, JavaScript, HTML, networks, and many other technologies. You can follow him @DanielKhoshnoud[7]

Rob Stewart - started programming in high school on a Commodore VIC-20. He taught himself BASIC, 6502 machine code, Forth, C, C++, JavaScript, Python, and other programming languages. (He also took a course on Fortran in college.) He has been using C++ for 30 years for things like cockpit simulators, commercial real estate tools, web browsing accelerators, computer desktop alternatives, financial trading, network communications, and more, while working for the US Air Force, startups, and Susquehanna International Group. He actually writes documentation (!) for his own libraries and has helped with the documentation for numerous Boost libraries. He has helped with several well-known C++ books. He has taught C++ classes and mentored many developers. He was a founding member of the Boost Steering Committee. He and his wife of 33 years have nine children.

[7] https://twitter.com/DanielKhoshnoud

Foreword

If you've ever asked "what's in C++17 and what does it mean for me and my code?" — and I hope you have — then this book is for you.

Now that the C++ standard is being released regularly every three years, one of the challenges we have as a community is learning and absorbing the new features that are being regularly added to the standard language and library. That means not only knowing what those features are, but also how to use them effectively to solve problems. Bartlomiej Filipek does a great job of this by not just listing the features, but explaining each of them with examples, including a whole Part 3 of the book about how to apply key new C++17 features to modernize and improve existing code — everything from upgrading `enable_if` to the new `if constexpr`, to refactoring code by applying the new `optional` and `variant` vocabulary types, to writing parallel code using the new standard parallel algorithms. In each case, the result is cleaner code that's often also significantly faster too.

The point of new features isn't just to know about them for their own sake, but to know about how they can let us express our intent more clearly and directly than ever in our C++ code. That ability to directly "say what we mean" to express our intent, or to express "what" we want to achieve rather than sometimes-tortuous details of "how" to achieve it through indirect mechanisms, is the primary thing that determines how clean and writable and readable — and correct — our code will be. For C++ programmers working on real-world projects using reasonably up-to-date C++ compilers, C++17 is where it's at in the industry today for writing robust production code. Knowing what's in C++17 and how to use it well is an important tool that will elevate your day-to-day coding, and more likely than not reduce your day-to-day maintenance and debugging chores.

If you're one of the many who have enjoyed Barteks's blog (bfilipek.com, frequently cited at isocpp.org), you'll certainly also enjoy this entertaining and informative book. And if you haven't enjoyed his blog yet, you should check it out too… and then enjoy the book.

Herb Sutter, herbsutter.com

Preface

After the long-awaited C++11, the C++ Committee has made changes to the standardisation process, and we can now expect a new language standard every three years. In 2014 the ISO Committee delivered C++14. Now it's time for C++17, which was published at the end of 2017. As I am writing these words, in the middle of 2019, the C++20 draft is feature ready and prepared for the final review process.

As you can see, the language and the Standard Library evolves quite fast! Since 2011 you've got a set of new library modules and language features every three years. Thus, staying up to date with the whole state of the language has become quite a challenging task, and that is why this book will help you.

This book describes all the significant changes in C++17 and will give you the essential knowledge to stay current with the latest features. What's more, each section contains lots of practical examples and also compiler-specific notes to provide you with a more comfortable start.

It's a pleasure for me to write about new and exciting things in the language and I hope you'll have fun discovering C++17 as well!

Best regards,

Bartek

About the Book

C++11 was a major update for the language. With all the modern features like lambdas, constexpr, variadic templates, threading, range-based for loops, smart pointers and many more powerful elements, it signalled enormous progress for the language. Even now, in 2019, many teams struggle to modernise their projects to leverage all the modern features. Later there was a minor update - C++14, which improved some things from the previous Standard and added a few smaller elements.

Although C++17 is not as big as C++11, it's larger than C++14 and brings many exciting additions and improvements. And this book will guide through all of them!

The book brings you exclusive content about C++17 and draws from the experience of many articles that have appeared at bfilipek.com. The material was rewritten from the ground-up and updated with the latest information. All of that equipped with lots of new examples and practical tips. Additionally, the book provides insight into the current implementation status, compiler support, performance issues and other relevant knowledge to boost your current projects.

Who This Book is For

This book is intended for all C++ developers who have at least essential experience with C++11/14.

The principal aim of the book is to equip you with practical knowledge about C++17. After reading the book, you'll be able to move past C++11 and leverage the latest C++ techniques in your day to day tasks.

Please don't worry if you're not an expert in C++11/14. This book provides the necessary background, so you'll get the information in a proper context.

Overall Structure of the Book

C++17 brings a lot of changes to the language and the Standard Library. In this book, all the new features were categorised into a few segments, so that they are easier to comprehend.

As a result, the book has the following sections:

- Part One - C++17 Language Features
  - Fixes and Deprecation
  - Language Clarification
  - General Language Features
  - Templates
  - Attributes
- Part Two - C++17 The Standard Library (full version only)
  - std::optional
  - std::variant
  - std::any
  - std::string_view
  - String Operations
  - Filesystem
  - Parallel STL
  - Other Changes
- Part Three - More Examples and Use Cases (full version only)
  - Refactoring with std::optional and std::variant
  - Enforcing Code Contracts With [[nodiscard]]
  - Replacing enable_if with ifconstexpr
  - How to Parallelise CSV Reader
- Appendix A - Compiler Support
- Appendix B - Resources and Links

Part One, about the language features, is shorter and will give you a quick run over the most significant changes. You can read it in any order you like.

Part Two (full version only), describes a set of new types and utilities that were added to the Standard Library. The helper types create a potential new vocabulary for C++ code: like when you use `optional`, `any`, `variant` or `string_view`. And what's more, you have new powerful capabilities, especially in the form of parallel algorithms and the standard filesystem. A lot of examples in this part will use many other features from C++17.

Part Three (full version only) brings together all of the changes in the language and shows examples where a lot of new features are used alongside. You'll see discussions about refactoring, simplifying code with new template techniques or working with parallel STL and the filesystem. While the first and the second part can also be used as a reference for individual changes, the third part shows more of larger C++17 patterns that join many features.

A considerable advantage of the book is the fact that with each new feature you'll get information about the compiler support and the current implementation status. That way you'll be able to check if a particular version of the most popular compilers (MSVC, GCC or Clang) implements it or not. The book also gives practical hints on how to apply new techniques in your current codebase.

Reader Feedback

If you spot an error, a typo, a grammar mistake... or anything else (especially logical issues!) that should be corrected, then please send your feedback to bartlomiej.filipek AT bfilipek.com.

You can also use those two places to leave your feedback:

- Leanpub Book's Feedback Page[8]
- GoodReads Book's Page[9]

Example Code

You can find the ZIP package with all the example on the book's website:

cppindetail.com/data/cpp17indetail.zip[10]

[8] https://leanpub.com/cpp17indetail/feedback
[9] https://www.goodreads.com/book/show/41447221-c-17-in-detail
[10] https://www.cppindetail.com/data/cpp17indetail.zip

The same ZIP package should also be attached with the ebook.

Many examples in the book are relatively short. You can copy and paste the lines into your favourite compiler/IDE and then run the code snippet.

Code License

The code for the book is available under the Creative Commons License.

Compiling

To use C++17 make sure you provide a proper flag for your compiler:

- for GCC (at least 7.1 or 8.0 or newer): use `-std=c++17` or `-std=c++2a`
- for Clang (at least 4.0 or newer): use `-std=c++17` or `-std=c++2a`
- for MSVC (Visual Studio 2017 or newer): use `/std:c++17` or `/std:c++latest` in `project options -> C/C++ -> Language -> C++ Language Standard`

Formatting

The code is presented in a monospace font, similarly to the following example:

For longer examples with a corresponding `cpp` file:

ChapterABC/example_one.cpp

```
#include <iostream>

int main() {
    std::string text = "Hello World";
    std::cout << text << '\n';
}
```

Or shorter snippets (without a corresponding file):

```
int foo() {
    return std::clamp(100, 1000, 1001);
}
```

Snippets of longer programs were usually shortened to present only the core mechanics. In that case, you'll find their full version in the separate ZIP package that comes with the book.

The corresponding file for the code snippet is mentioned in the title above the frame:

`Chapter ABC/example_one.cpp`

Usually, source code uses full type names with namespaces, like `std::string`, `std::clamp`, `std::pmr`. However, to make code compact and present it nicely on a book page the namespaces sometimes might be removed, so they don't use space. Also, to avoid line wrapping, longer lines might be manually split into two. In some case, the code in the book might skip `include` statements.

Syntax Highlighting Limitations

The current version of the book might show some Pygments syntax highlighting limitations.

For example:

- `if constexpr` - Link to Pygments issue: #1432 - C++ if constexpr not recognized (C++17)[11]
- The first method of a class is not highlighted - #1084 - First method of class not highlighted in C++[12]
- Template method is not highlighted #1434 - C++ lexer doesn't recognize function if return type is templated[13]
- Modern C++ attributes are sometimes not recognised properly

Other issues for C++ and Pygments: issues C++[14].

[11] https://bitbucket.org/birkenfeld/pygments-main/issues/1432/c-if-constexpr-not-recognized-c-17
[12] https://bitbucket.org/birkenfeld/pygments-main/issues/1084/first-method-of-class-not-highlighted-in-c
[13] https://bitbucket.org/birkenfeld/pygments-main/issues/1434/c-lexer-doesnt-recognize-function-if
[14] https://bitbucket.org/birkenfeld/pygments-main/issues?q=c%2B%2B

Online Compilers

Instead of creating local projects to play with the code samples, you can also leverage online compilers. They offer a basic text editor and usually allow you to compile only one source file (the code that you edit). They are convenient if you want to play with code samples and check the results using various compilers.

For example, many of the code samples for this book were created using Coliru Online and Wandbox compilers and then adapted for the book.

Here's a list of some of the useful services:

- Coliru[15] - uses GCC 8.2.0 (as of July 2019), offers link sharing and a basic text editor, it's simple but very effective.
- Wandbox[16] - offers a lot of compilers, including most Clang and GCC versions, can use boost libraries; offers link sharing and multiple file compilation.
- Compiler Explorer[17] - offers many compilers, shows compiler output, can execute the code.
- CppBench[18] - runs simple C++ performance tests (using google benchmark library).
- C++ Insights[19] - a Clang-based tool for source to source transformation. It shows how the compiler sees the code, for example by expanding lambdas, auto, structured bindings or range-based for loops.

There's also a helpful list of online compilers gathered on this website: List of Online C++ Compilers[20].

[15] http://coliru.stacked-crooked.com/
[16] https://wandbox.org/
[17] https://gcc.godbolt.org/
[18] http://quick-bench.com/
[19] https://cppinsights.io/
[20] https://arnemertz.github.io/online-compilers/

Part 1 - Language Features

We can say that C++ comes in two parts: The Language and The Standard Library. The first element, The Language, focuses on the expressive code and conscience syntax. The second element gives you tools, utilities and algorithms. For example, in C++11, we got lambdas that simplified writing short function objects. C++14 allowed 'auto' type deduction for function return types which also shorten code and simplified templated code.

C++17, as a major update to the Standard, brings many amazing language elements that generally, make the language clearer and more straightforward. For instance, you can reduce the need to use `enable_if` and tag dispatching techniques by leveraging `if constexpr`. You can treat tuples like first class language citizens thanks to structured bindings, rely on and understand expression evaluation order, write code that naturally uses copy-elision mechanism, and much, much more!

In this part you'll learn:

- What was removed from the language and what is deprecated
- How the language is more precise: for example, thanks to expression evaluation order guarantees
- What are new features of templates: like `if constexpr` or fold expressions
- What are the new standard attributes
- How you can write cleaner and more expressive code thanks to structured binding, inline variables, compile-time if or template argument deduction for classes

1. Quick Start

To make you more curious about the new Standard, below, there are a few code samples that present combined language features.

Don't worry if you find the examples confusing or too complicated. All the new features are individually explained in depth in the coming chapters.

Working With Maps

Part I/demo_map.cpp

```
1  #include <iostream>
2  #include <map>
3
4  int main() {
5      std::map<std::string, int> mapUsersAge { { "Alex", 45 }, { "John", 25 } };
6
7      std::map mapCopy{mapUsersAge};
8
9      if (auto [iter, wasAdded] = mapCopy.insert_or_assign("John", 26); !wasAdded)
10         std::cout << iter->first << " reassigned...\n";
11
12     for (const auto& [key, value] : mapCopy)
13         std::cout << key << ", " << value << '\n';
14 }
```

The code will output:

```
John reassigned...
Alex, 45
John, 26
```

The above example uses the following features:

- Line 7: Template Argument Deduction for Class Templates - `mapCopy` type is deduced from the type of `mapUsersAge`. No need to declare `std::map<std::string, int> mapCopy{...}`.

- Line 9: New inserting member function for maps - `insert_or_assign`.
- Line 9: Structured Bindings - captures a returned pair from `insert_or_assign` into separate names.
- Line 9: init if statement - `iter` and `wasAdded` are visible only in the scope of the surrounding `if` statement.
- Line 12: Structured Bindings inside a range-based for loop - we can iterate using `key` and `value` rather than `pair.first` and `pair.second`.

Debug Printing

Part I/demo_print.cpp

```
1  #include <iostream>
2
3  template<typename T> void linePrinter(const T& x)  {
4      if constexpr (std::is_integral_v<T>)
5          std::cout << "num: " << x << '\n';
6      else if constexpr (std::is_floating_point_v<T>) {
7          const auto frac = x - static_cast<long>(x);
8          std::cout << "flt: " << x << ", frac " << frac << '\n';
9      }
10     else if constexpr(std::is_pointer_v<T>) {
11         std::cout << "ptr, ";
12         linePrinter(*x);
13     }
14     else
15         std::cout << x << '\n';
16 }
17
18 template<typename ... Args> void printWithInfo(Args ... args) {
19     (linePrinter(args), ...); // fold expression over the comma operator
20 }
21
22 int main () {
23     int i = 10;
24     float f = 2.56f;
25     printWithInfo(&i, &f, 30);
26 }
```

The code will output:

```
ptr, num: 10
ptr, flt: 2.56, frac 0.56
num: 30
```

Here you can see the following features:

- Line 4, 6, 10: `if constexpr` - to discard code at compile-time, used to match the template parameter.
- Line 4, 6, 10: `_v` variable templates for type traits - no need to write `std::trait_name<T>::value`.
- Line 19: Fold Expressions inside `printWithInfo`. This feature simplifies variadic templates. In the example, we invoke `linePrinter()` over all input arguments.

Let's Start!

The code you've seen so far is just the tip of the iceberg! Continue reading to see much more: fixes in the language, clarifications, removed things (like `auto_ptr`), and of course the new stuff: `constexpr` lambda, `if constexpr`, fold expressions, structured bindings, `template<auto>`, inline variables, template argument deduction for class templates and much more!

2. Removed or Fixed Language Features

The C++17 Standard contains over 1600 pages, growing over 200 pages compared to C++14[1]! Fortunately, the language specification was cleaned up in a few places, and some old or potentially harmful features could be cleared out. This short chapter lists several language elements that were removed or fixed. See the Removed And Deprecated Library Features Chapter for a list of changes in the Standard Library.

In this chapter, you'll learn:

- What was removed from the language like the `register` keyword or `operator++` for `bool`
- What was fixed, notably the auto type deduction with brace initialisation.
- Other improvements like for `static_assert` and range-based for loops.

[1]For example the draft from July 2017 N4687 compared to C++14 draft N4140

Removed Elements

One of the core concepts behind each iteration of C++ is its compatibility with previous versions. We'd like to have new things in the language, but at the same time, our old projects should still compile. However, sometimes, there's a chance to remove parts that are wrong or rarely used.

This section briefly explains what was removed from the Standard.

Removing the `register` Keyword

The `register` keyword was deprecated in 2011 (C++11), and it has had no meaning since then. It was removed in C++17. The keyword is reserved and might be repurposed in future revisions of the Standard (for example `auto` keyword was reused and now is an entirely new and powerful feature).

If you use `register` to declare a variable:

```
register int a;
```

You might get the following warning (GCC8.1 below)

```
warning: ISO C++17 does not allow 'register' storage class specifier
```

or error in Clang (Clang 7.0)

```
error: ISO C++17 does not allow 'register' storage class specifier
```

Extra Info

The change was proposed in: P0001R1[2].

[2] https://wg21.link/p0001r1

Removing Deprecated `operator++(bool)`

The increment operator for `bool` has been already deprecated for a very long time! The committee recommended against its use back in 1998 (C++98), but they only now finally agreed to remove it from the language. Note that `operator--` was never enabled for `bool`.

If you try to write the following code:

```
bool b;
b++;
```

You should get a similar error like this from GCC (GCC 8.1):

```
error: use of an operand of type 'bool' in 'operator++' is forbidden in C++17
```

Extra Info

The change was proposed in: P0002R1[3].

Removing Deprecated Exception Specifications

In C++17, exception specification will be part of the type system (as discussed in the next chapter about Language Clarification). However, the standard contains old and deprecated exception specification that appeared to be impractical and unused.

For example:

```
void fooThrowsInt(int a) throw(int) {
   printf_s("can throw ints\n");
   if (a == 0)
      throw 1;
}
```

Pay special attention to that `throw(int)` part.

The above code has been deprecated since C++11. The only practical exception declaration is `throw()` which means - this code won't throw anything. Since C++11 it's been advised to use `noexcept`.

[3] https://wg21.link/p0002r1

For example in clang 4.0 you'll get the following error:

```
error: ISO C++1z does not allow dynamic exception specifications
[-Wdynamic-exception-spec] note: use 'noexcept(false)' instead
```

Extra Info

The change was proposed in: P0003R5[4].

Removing Trigraphs

Trigraphs are special character sequences that could be used when a system doesn't support 7-bit ASCII (like ISO 646). For example `??=` generated #, `??-` produced ~. (All of C++'s basic source character set fits in 7-bit ASCII). Today, trigraphs are rarely used, and by removing them from the translation phase, the compilation process can be more straightforward. See a table below with all the trighraps that were declared until C++17:

Trigraph	Replacement
??=	#
??(	[
??<	{
??/	\
??)	]
??>	}
??'	^
??!	|
??-	~

Extra Info

The change was proposed in: N4086[5].

[4] http://wg21.link/p0003r5
[5] https://wg21.link/n4086

Fixes

We can argue what is a fix in a language standard and what is not. Below there are three things that might look like a fix for something that was missing or not working in the previous rules.

New auto rules for direct-list-initialisation

Since C++11 there's been a strange problem where:

```
auto x { 1 };
```

Is deduced as `std::initializer_list<int>`. Such behaviour is not intuitive as in most cases you should expect it to work like `int x { 1 };`.

Brace initialisation is the preferred pattern in modern C++, but such exceptions make the feature weaker.

With the new Standard, we can fix this so that it will deduce `int`.

To make this happen, we need to understand two ways of initialisation - copy and direct:

```
// foo() is a function that returns some Type by value
auto x = foo();    // copy-initialisation
auto x{foo()};     // direct-initialisation, initializes an
                   // initializer_list (until C++17)

int x = foo();     // copy-initialisation
int x{foo()};      // direct-initialisation
```

For the direct initialisation, C++17 introduces new rules:

- For a braced-init-list with a single element, auto deduction will deduce from that entry.
- For a braced-init-list with more than one element, auto deduction will be ill-formed.

For example:

```
auto x1 = { 1, 2 };   // decltype(x1) is std::initializer_list<int>
auto x2 = { 1, 2.0 }; // error: cannot deduce element type
auto x3{ 1, 2 };      // error: not a single element
auto x4 = { 3 };      // decltype(x4) is std::initializer_list<int>
auto x5{ 3 };         // decltype(x5) is int
```

Extra Info

The change was proposed in: N3922[6] and N3681[7]. The compilers fixed this issue quite early, as the improvement is available in GCC 5.0 (Mid 2015), Clang 3.8 (Early 2016) and MSVC 2015 (Mid 2015). Much earlier than C++17 was approved.

`static_assert` With no Message

This feature adds a new overload for `static_assert`. It enables you to have the condition inside `static_assert` without passing the message.

It will be compatible with other asserts like `BOOST_STATIC_ASSERT`. Programmers with boost experience will now have no trouble switching to C++17 `static_assert`.

```
static_assert(std::is_arithmetic_v<T>, "T must be arithmetic");
static_assert(std::is_arithmetic_v<T>); // no message needed since C++17
```

In many cases, the condition you check is expressive enough and doesn't need to be mentioned in the message string.

Extra Info

The change was proposed in: N3928[8].

[6] http://wg21.link/n3922
[7] http://wg21.link/n3681
[8] https://wg21.link/n3928

Different `begin` and `end` Types in Range-Based For Loop

C++11 added range-based for loops:

```
for (for-range-declaration : for-range-initializer)
    statement;
```

According to the C++14 standard that loop is equivalent to the following code:

```
auto && __range = for-range-initializer;
for ( auto __begin = begin-expr, __end = end-expr;
          __begin != __end;
          ++__begin ) {
    for-range-declaration = *__begin;
    statement;
}
```

As you can see, `__begin` and `__end` have the same type. This works nicely but is not scalable enough. For example, you might want to iterate until some sentinel value with a different type than the start of the range.

In C++17 range-based for loops are defined as equivalent to the following code:

```
auto && __range = for-range-initializer;
auto __begin = begin-expr;
auto __end = end-expr;
for ( ; __begin != __end; ++__begin ) {
    for-range-declaration = *__begin;
    statement;
}
```

The types of `__begin` and `__end` might be different; only the comparison operator is required. That change has no effect on existing for loops but it provides more options for libraries. For example, this little change allows Range TS (and Ranges in C++20) to work with the range-based for loop.

Extra Info

The change was proposed in: P0184R0[9].

[9] https://wg21.link/p0184r0

Compiler Support

Feature	GCC	Clang	MSVC	C++ Builder
Removing `register` keyword	7.0	3.8	VS 2017 15.3	10.3 Rio
Remove Deprecated `operator++`(bool)	7.0	3.8	VS 2017 15.3	10.3 Rio
Removing Deprecated Exception Specifications	7.0	4.0	VS 2017 15.5	10.3 Rio
Removing trigraphs	5.1	3.5	VS 2010	10.3 Rio
New auto rules for direct-list-initialisation	5.0	3.8	VS 2015	10.3 Rio
`static_assert` with no message	6.0	2.5	VS 2017	10.3 Rio
Different begin and end types in range-based for	6.0	3.6	VS 2017	10.3 Rio

3. Language Clarification

C++ is a challenging language to learn and fully understand, and some parts might be confusing for programmers. One of the reasons for the lack of clarity might be the freedom given to the implementation/compiler. For example, some parts of the language are left vague to allow for more aggressive optimisations. Other difficulties can arise from the requirement to be compatible with C. C++17 addresses some of the most common "holes" in the language.

In this chapter, you'll learn:

- What Evaluation Order is and why it might generate unexpected results
- Copy elision guarantees in the language
- Exceptions specifications as part of the type system
- Memory allocations for (over)aligned data

Stricter Expression Evaluation Order

Until C++17 the language hasn't specified any evaluation order for function parameters. **Period**.

For example, that's why in C++14 `make_unique` is not just syntactic sugar, but it guarantees memory safety:

Consider the following examples:

```
foo(unique_ptr<T>(new T), otherFunction()); // first case
```

And with explicit `new`.

```
foo(make_unique<T>(), otherFunction()); // second case
```

Considering the first case, in C++14, we only know that `new T` is guaranteed to happen before the `unique_ptr` construction, but that's all. For example, `new T` might be called first, then `otherFunction()`, and then the constructor `unique_ptr` is invoked.

For such evaluation order, when `otherFunction()` throws, then `new T` generates a leak (as the unique pointer is not yet created).

When you use `make_unique`, as in the second case, the leak is not possible as you wrap memory allocation and creation of unique pointer in one call.

C++17 addresses the issue shown in the first case. Now, the evaluation order of function arguments is "practical" and predictable. In our example, the compiler won't be allowed to call `otherFunction()` before the expression `unique_ptr<T>(new T)` is fully evaluated.

The Changes

In an expression:

```
f(a, b, c);
```

The order of evaluation of `a`, `b`, `c` is still unspecified, but any parameter is fully evaluated before the next one is started. It's especially crucial for complex expressions like this:

```
f(a(x), b, c(y));
```

If the compiler chooses to evaluate `a(x)` first, then it must evaluate `x` before processing `b`, `c(y)` or `y`.

This guarantee fixes the problem with `make_unique` vs `unique_ptr<T>(new T())`. A given function argument must be fully evaluated before other arguments are evaluated.

Consider the following case:

Chapter Clarification/chain_order.cpp

```
#include <iostream>

class Query {
public:
    Query& addInt(int i) {
        std::cout << "addInt: " << i << '\n';
        return *this;
    }

    Query& addFloat(float f) {
        std::cout << "addFloat: " << f << '\n';
        return *this;
    }
};

float computeFloat() {
    std::cout << "computing float... \n";
    return 10.1f;
}

float computeInt() {
    std::cout << "computing int... \n";
    return 8;
}

int main() {
  Query q;
  q.addFloat(computeFloat()).addInt(computeInt());
}
```

You probably expect that using C++14 `computeInt()` happens after `addFloat`. Unfortunately, that might not be the case. For instance here's an output from GCC 4.7.3:

```
computing int...
computing float...
addFloat: 10.1
addInt: 8
```

The chaining of functions is already specified to work from left to right (thus `addInt()` happens after `addFloat()`), but the order of evaluation of the inner expressions can differ. To be precise:

> The expressions are indeterminately sequenced with respect to each other.

With C++17, function chaining will work as expected when they contain inner expressions, i.e., they are evaluated from left to right:

In the expression:

```
a(expA).b(expB).c(expC)
```

`expA` is evaluated before calling `b()`.

Compiling the previous example with a conformant C++17 compiler, yields the following result:

```
computing float...
addFloat: 10.1
computing int...
addInt: 8
```

Another result of this change is that when using operator overloading, the order of evaluation is determined by the order associated with the corresponding built-in operator.

For example:

```
std::cout << a() << b() << c();
```

The above code contains operator overloading and expands to the following function notation:

```
operator<<(operator<<(operator<<(std::cout, a()), b()), c());
```

Before C++17, `a()`, `b()` and `c()` could be evaluated in any order. Now, in C++17, `a()` will be evaluated first, then `b()` and then `c()`.

Here are more rules described in the paper P0145R3[1]:

> the following expressions are evaluated in the order a, then b:
>
> 1. `a.b`
> 2. `a->b`
> 3. `a->*b`
> 4. `a(b1, b2, b3) // b1, b2, b3 - in any order`
> 5. `b @= a // '@' means any operator`
> 6. `a[b]`
> 7. `a << b`
> 8. `a >> b`

If you're not sure how your code might be evaluated, then it's better to make it simple and split it into several clear statements. You can find some guides in the Core C++ Guidelines, for example ES.44[2] and ES.44[3].

Extra Info

The change was proposed in: P0145R3[4].

[1] https://wg21.link/p0145r3
[2] http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-order
[3] http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#es44-dont-depend-on-order-of-evaluation-of-function-arguments
[4] https://wg21.link/p0145r3

Guaranteed Copy Elision

Copy Elision is a common optimisation that avoids creating unnecessary temporary objects.

For example:

Chapter Clarification/copy_elision.cpp

```
#include <iostream>

struct Test {
    Test() { std::cout << "Test::Test\n"; }
    Test(const Test&) { std::cout << "Test(const Test&)\n"; }
    Test(Test&&) { std::cout << "Test(Test&&)\n"; }
    ~Test() { std::cout << "~Test\n"; }
};

Test Create() {
    return Test();
}

int main() {
    auto n = Create();
}
```

In the above call, you might assume a temporary copy is used - to store the return value of `Create`. In C++14, most compilers recognise that the temporary object can be optimised easily, and they can create `n` "directly" from the call of `Create()`. So you'll probably see the following output:

```
Test::Test // create n
~Test // destroy n when main finishes
```

In its basic form, the copy elision optimisation is called Return Value Optimisation (**RVO**).

As an experiment, in GCC you can add a compiler flag `-fno-elide-constructors` and use `-std=c++14` (or some earlier language standard). In that case you'll see a different output:

```
// compiled as "g++ CopyElision.cpp -std=c++14 -fno-elide-constructors"
Test::Test
Test(Test&&)
~Test
Test(Test&&)
~Test
~Test
```

In this case, we have two extra copies that the compiler uses to pass the return value into `n`;

Compilers are even smarter, and they can elide in cases when you return a named object - it's called **Named Return Value Optimisation - NRVO**:

```
Test Create() {
    Test t;
    // several instruction to initialize 't'...
    return t;
}

auto n = Create(); // temporary will be usually elided
```

Currently, the Standard allows eliding in cases like:

- When a temporary object is used to initialise another object (including the object returned by a function, or the exception object created by a throw-expression)
- When a variable that is about to go out of scope is returned or thrown
- When an exception is caught by value

However, it's up to the compiler/implementation to elide or not. In practice, all the constructors' definitions are required.

With C++17, we get clear rules on when elision has to happen, and thus constructors might be entirely omitted. In fact, instead of eliding the copies the compiler defers the "materialisation" of an object.

Why might this be useful?

- To allow returning objects that are not movable/copyable - because we could now skip copy/move constructors
- To improve code portability since every conformant compiler supports the same rule

- To support the *"return by value"* pattern rather than using output arguments
- To improve performance

Below you can see an example with a non-movable/non-copyable type, based on P0135R0:

Chapter Clarification/copy_elision_non_moveable.cpp

```
struct NonMoveable {
    NonMoveable(int x) : v(x) { }
    NonMoveable(const NonMoveable&) = delete;
    NonMoveable(NonMoveable&&) = delete;

    std::array<int, 1024> arr;
    int v;
};

NonMoveable make(int val) {
    if (val > 0)
        return NonMoveable(val);

    return NonMoveable(-val);
}

int main() {
    auto largeNonMoveableObj = make(90); // construct the object
    return largeNonMoveableObj.v;
}
```

The above code wouldn't compile under C++14 as it lacks copy and move constructors. But with C++17 the constructors are not required - because the object `largeNonMovableObj` will be constructed in place.

Please notice that you can also use many return statements in one function and copy elision will still work.

Moreover, it's important to remember, that in C++17 copy elision works only for unnamed temporary objects, and Named RVO is not mandatory.

To understand how mandatory copy elision/deferred temporary materialisation is defined in the C++ Standard, we must understand value categories which are covered in the next section.

Updated Value Categories

In C++98/03, we had two basic categories of expressions:

- `lvalue` - an expression that can appear on the left-hand side of an assignment
- `rvalue` - an expression that can appear only on the right-hand side of an assignment

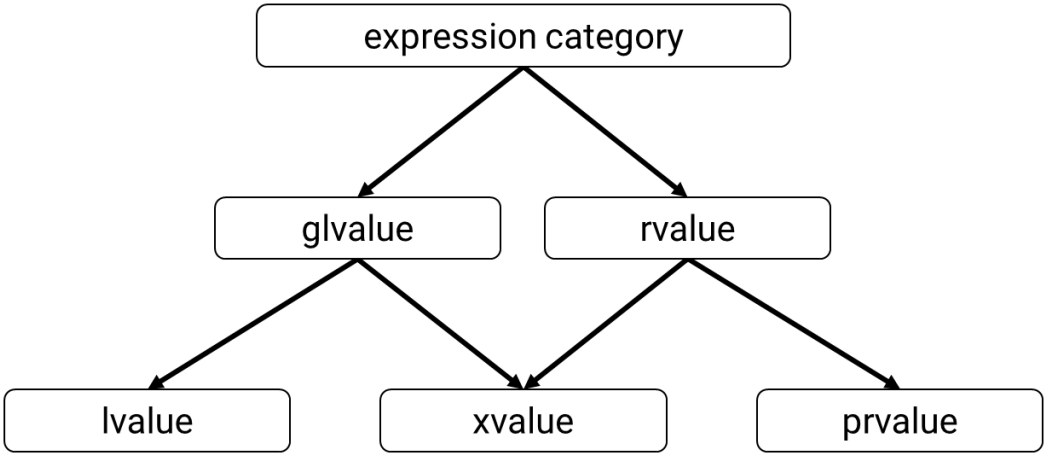
C++11 extended this taxonomy (due to the move semantics), with three more categories:

- `xvalue` - an eXpiring `lvalue`
- `prvalue` - a pure `rvalue`, an `xvalue`, a temporary object or subobject, or a value that is not associated with an object.
- `glvalue` - a generalised `lvalue`, which is an `lvalue` or an `xvalue`

Examples:

```
std::string str;
str;            // lvalue
42;             // prvalue
str + "10"      // prvalue
std::move(str); // xvalue
```

Here's a diagram that shows how the categories are related:



Value Categories

There are three core categories (below with colloquial "definitions"):

- `lvalue` - an expression that has an identity, and which we can take the address of
- `xvalue` - "eXpiring `lvalue`" - an object that we can move from, which we can reuse. Usually, its lifetime ends soon
- `prvalue` - pure `rvalue` - something without a name, which we cannot take the address of, we can move from such expression

To support Copy Elision, the authors of the proposal provided the updated definitions of `glvalue` and `prvalue`. From the Standard[5]:

> - `glvalue` - A `glvalue` is an expression whose evaluation computes the location of an object, bit-field, or function
> - `prvalue` - A `prvalue` is an expression whose evaluation initialises an object, bit-field, or operand of an operator, as specified by the context in which it appears

For example:

```
class X { int a; };
X{10}   // this expression is prvalue
X x;    // x is lvalue
x.a     // it's lvalue (location)
```

In short: `prvalues` perform initialisation, `glvalues` describe locations. The C++17 Standard specifies that when there's a `prvalue` initialising some `glvalue`, then there's no need to create a temporary and we can defer its materialisation.

In C++17 Copy Elision/Deferred Temporary Materialization happens when:

- in initialisation of an object from a `prvalue`: `Type t = T()`
- in a function call where the function returns a `prvalue` - like in our examples.

Extra Info

The change was proposed in: P0135R0[6](reasoning) - and P0135R1[7](wording).

[5] https://timsong-cpp.github.io/cppwp/n4659/basic.lval
[6] https://wg21.link/p0135r0
[7] https://wg21.link/p0135r1

Dynamic Memory Allocation for Over-Aligned Data

Embedded environments, kernel, drivers, game development and other areas might require a non-default alignment for memory allocations. Complying those requirements might improve the performance or satisfy some hardware interface.

For example, to perform geometric data processing using SIMD[8] instructions, you might need 16-byte or 32-byte alignment for a structure that holds 3D coordinates:

```
struct alignas(32) Vec3d { // alignas is available since C++11
    double x, y, z;
};
auto pVectors = new Vec3d[1000];
```

`Vec3d` holds `double` fields, and usually, its natural alignment should be 8 bytes. Now, with `alignas` keyword, we change this alignment to 32. This approach allows the compiler to fit the objects into SIMD registers like AVX (256-bit-wide registers).

Unfortunately, in C++11/14, you have no guarantee how the memory will be aligned after `new[]`. Often, you have to use routines like `std::aligned_alloc()` or MSVC's `_aligned_malloc()` to be sure the alignment is preserved. That's not ideal as it's not working easily with smart pointers and also makes memory management visible in the code.

C++17 fixes that hole by introducing new memory allocation function overloads for `new()` and `delete()` with the `align_val_t` parameter. Example function signatures below[9]:

```
void* operator new(size_t, align_val_t);
void operator delete(void*, size_t, align_val_t);
```

The Standard also defines `__STDCPP_DEFAULT_NEW_ALIGNMENT__` macro that specifies the default alignment for dynamic memory allocations. On common platforms, Clang, GCC and MSVC specify it as 16 bytes.

Now, in C++17, when you allocate:

```
auto pVectors = new Vec3d[1000];
```

[8] Single Instruction, Multiple Data, for example, SSE2, AVX, see https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions
[9] See all 22 `new()` overloads at https://en.cppreference.com/w/cpp/memory/new/operator_new

The alignment for `Vec3d` is larger than `__STDCPP_DEFAULT_NEW_ALIGNMENT__`, and thus the compiler will select the overloads with the `align_val_t` parameter.

In Clang and GCC you can control the default alignment by using the `fnew-alignment` switch (see Clang's documentation[10]). The MSVC compiler exposes the `/Zc:alignedNew`[11] flag that turns the feature on or off.

We can also provide custom implementation, have a look:

Chapter Clarification/aligned_new.cpp

```
void* operator new(std::size_t size, std::align_val_t align) {
#if defined(_WIN32) || defined(__CYGWIN__)
    auto ptr = _aligned_malloc(size, static_cast<std::size_t>(align));
#else
    auto ptr = std::aligned_alloc(static_cast<std::size_t>(align), size);
#endif

    if (!ptr) throw std::bad_alloc{};

    std::cout << "new: " << size << ", align: "
              << static_cast<std::size_t>(align) << ", ptr: " << ptr << '\n';

    return ptr;
}

void operator delete(void* ptr, std::size_t size, std::align_val_t algn) noexcept {
    std::cout << "delete: " << size << ", align: "
              << static_cast<std::size_t>(algn) << ", ptr : " << ptr << '\n';
#if defined(_WIN32) || defined(__CYGWIN__)
    _aligned_free(ptr);
#else
    std::free(ptr);
#endif
}

void operator delete(void* ptr, std::align_val_t algn) noexcept { ... } // hidden
```

The code uses `_aligned_malloc()` and `_aligned_free()` for the Windows version[12]. It's because the Windows platform uses different allocation mechanisms for over-aligned

[10] https://clang.llvm.org/docs/ClangCommandLineReference.html#cmdoption-clang-fnew-alignment
[11] https://docs.microsoft.com/en-us/cpp/build/reference/zc-alignednew?view=vs-2019
[12] this applies to MSVC, MinGW, Clang on Windows or Cygwin.

data, and that's why `std::free()` wouldn't release the memory correctly. On other platforms that conform with C11 you can try using `std::aligned_alloc()`, as since C++17 the Standard is based on the C11 specification. In that context `free()` can delete the aligned memory.

The new functionality can significantly improve the code, and you can now hold the aligned objects in standard containers without writing custom allocators, or custom deleters for smart pointers.

For example:

Chapter Clarification/aligned_new.cpp

```
std::vector<Vec3d> vec;
vec.push_back({});
vec.push_back({});
vec.push_back({});
assert(reinterpret_cast<uintptr_t>(vec.data()) % alignof(Vec3d) == 0);
```

When executed, our replaced allocation functions might log the following output:

```
new: 32, align: 32, ptr: 000001F866625960
new: 64, align: 32, ptr: 000001F866625680
delete: 32, align: 32, ptr : 000001F866625960
new: 96, align: 32, ptr: 000001F866623EA0
delete: 64, align: 32, ptr : 000001F866625680
delete: 96, align: 32, ptr : 000001F866623EA0
```

The example allocates memory for a single entry, then deletes it and increases the size for the vector twice to make space for all three elements. At the end we check the pointer alignment to be sure it's aligned to 32 bytes.

You can read more about the experiments with the new functionality at New new() - The C++17's Alignment Parameter for Operator new()[13]. This blog post also shows the dangerous side when you try to ask for a non-standard alignment with placement new.

Extra Info

The change was proposed in: P0035[14].

[13] https://www.bfilipek.com/2019/08/newnew-align.html
[14] http://wg21.link/p0035

Exception Specifications in the Type System

Exception Specification for a function didn't use to belong to the type of the function, but now it will be part of it. You can now have two function overloads: one with `noexcept` and the second without it. See below:

Chapter Clarification/func_except_type.cpp

```
using TNoexceptVoidFunc = void (*)() noexcept;
void SimpleNoexceptCall(TNoexceptVoidFunc f) { f(); }

using TVoidFunc = void (*)();
void SimpleCall(TVoidFunc f) { f(); }

void fNoexcept() noexcept { }
void fRegular() { }

int main() {
    SimpleNoexceptCall(fNoexcept);
    //SimpleNoexceptCall(fRegular); // cannot convert

    SimpleCall(fNoexcept); // converts to regular function
    SimpleCall(fRegular);
}
```

A pointer to `noexcept` function can be converted to a pointer to a regular function (this also works for a pointer to a member function). But it's not possible the other way around (from a regular function pointer into a function pointer that is marked with `noexcept`).

One of the reasons for adding the feature is a chance to optimise the code better. If the compiler has a guarantee that a function won't throw, then it can generate faster code.

Also, as described in the previous chapter about Language Fixes, in C++17, the Exception Specification is cleaned up. Effectively, you can only use the `noexcept` specifier[15] for declaring that a function won't throw.

Extra Info

The change was proposed in: P0012R1[16].

[15] http://en.cppreference.com/w/cpp/language/noexcept_spec
[16] http://wg21.link/p0012r1

Compiler Support

Feature	GCC	Clang	MSVC	C++ Builder
Stricter expression evaluation order	7.0	4.0	VS 2017	10.3 Rio
Guaranteed copy elision	7.0	4.0	VS 2017 15.6	10.3 Rio
Exception specifications part of the type system	7.0	4.0	VS 2017 15.5	10.3 Rio
Dynamic memory allocation for over-aligned data	7.0	4.0	VS 2017 15.5	10.3 Rio

4. General Language Features

In this section of the book, we'll look at widespread improvements to the language that have the potential to make your code more compact and expressive. A perfect example of such a general feature is structured binding. Using that feature, you can leverage a comfortable syntax for tuples (and tuple-like expressions). Something easy in other languages like Python is now possible with C++17.

In this chapter, you'll learn:

- Structured bindings/Decomposition declarations
- How to provide Structured Binding interface for your custom classes
- Init-statement for if/switch
- Inline variables and their impact on header-only libraries
- Lambda expressions that might be used in a `constexpr` context
- How to properly wrap the `this` pointer in lambda expressions
- Simplified use of nested namespaces
- How to test for header existence with `__has_include` directive

Structured Binding Declarations

Do you often work with tuples or pairs?

If not, then you should probably start looking into those handy types. Tuples enable you to bundle data ad-hoc with excellent library support instead of creating small custom types. The language features like structured binding make the code even more expressive and concise.

Consider a function that returns two results in a pair:

```
std::pair<int, bool> InsertElement(int el) { ... }
```

You can write:

```
auto ret = InsertElement(...)
```

And then refer to `ret.first` or `ret.second`. However, referring to values as `.first` or `.second` is also not expressive - you can easily confuse the names, and it's hard to read. Alternatively you can leverage `std::tie` which will unpack the tuple/pair into local variables:

```
int index { 0 };
bool flag { false };
std::tie(index, flag) = InsertElement(10);
```

Such code might be useful when you work with `std::set::insert` which returns `std::pair`:

```
std::set<int> mySet;
std::set<int>::iterator iter;
bool inserted { false };

std::tie(iter, inserted) = mySet.insert(10);

if (inserted)
    std::cout << "Value was inserted\n";
```

As you see, such a simple pattern - returning several values from a function - requires several lines of code. Fortunately, C++17 makes it much simpler!

With C++17 you can write thw following:

```
std::set<int> mySet;

auto [iter, inserted] = mySet.insert(10);
```

Now, instead of `pair.first` and `pair.second`, you can use variables with concrete names. In addition, you have one line instead of three, and the code is easier to read. The code is also safer as `iter` and `inserted` are initialised in the expression.

Such syntax is called a *structured binding expression.*

The Syntax

The basic syntax for structured bindings is as follows:

```
auto [a, b, c, ...] = expression;
auto [a, b, c, ...] { expression };
auto [a, b, c, ...] ( expression );
```

The compiler introduces all identifiers from the `a, b, c, ...` list as names in the surrounding scope and binds them to sub-objects or elements of the object denoted by expression.

Behind the scenes, the compiler might generate the following **pseudo code**:

```
auto tempTuple = expression;
using a = tempTuple.first;
using b = tempTuple.second;
using c = tempTuple.third;
```

Conceptually, the expression is copied into a tuple-like object (`tempTuple`) with member variables that are exposed through `a`, `b` and `c`. However, the variables `a`, `b` and `c` are not references; they are aliases (or bindings) to the generated object member variables. The temporary object has a unique name assigned by the compiler.

For example:

```
std::pair a(0, 1.0f);
auto [x, y] = a;
```

`x` binds to `int` stored in the generated object that is a copy of `a`. And similarly, `y` binds to `float`.

Modifiers

Several modifiers can be used with structured bindings:

`const` modifiers:

```
const auto [a, b, c, ...] = expression;
```

References:

```
auto& [a, b, c, ...] = expression;
auto&& [a, b, c, ...] = expression;
```

For example:

```
std::pair a(0, 1.0f);
auto& [x, y] = a;
x = 10;  // write access
// a.first is now 10
```

In the example, `x` binds to the element in the generated object, that is a reference to `a`.

Now it's also quite easy to get a reference to a tuple member:

```
auto& [ refA, refB, refC, refD ] = myTuple;
```

You can also add `[[attribute]]` to structured bindings:

```
[[maybe_unused]] auto& [a, b, c, ...] = expression;
```

Structured Bindings or Decomposition Declaration?

You might have seen another name used for this feature: "decomposition declaration". During the standardisation process, both names were considered, but "structured bindings" was selected.

Structured Binding Limitations

There are several limitations related to structured bindings. They cannot be declared as `static` or `constexpr` and also they cannot be used in lambda captures. For example:

```
constexpr auto [x, y] = std::pair(0, 0);
// generates:
error: structured binding declaration cannot be 'constexpr'
```

It was also unclear about the linkage of the bindings. Compilers had a free choice to implement it (and thus some of them might allow capturing a structured binding in lambdas). Fortunately, those limitations might be removed due to C++20 proposal (already accepted): P1091: Extending structured bindings to be more like variable declarations[1].

Binding

Structured Binding is not only limited to tuples, we have three cases from which we can bind from:

1. If the initializer is an array:

```
// works with arrays:
double myArray[3] = { 1.0, 2.0, 3.0 };
auto [a, b, c] = myArray;
```

In this case, an array is copied into a temporary object, and `a`, `b` and `c` refers to copied elements from the array.

The number of identifiers must match the number of elements in the array.

2. If the initializer supports `std::tuple_size<>`, provides `get<N>()` and also exposes `std::tuple_element` functions:

```
std::pair myPair(0, 1.0f);
auto [a, b] = myPair; // binds myPair.first/second
```

In the above snippet, we bind to `myPair`. But this also means that you can provide support for your classes, assuming you add `get<N>` interface implementation. See an example in the later section.

[1] https://wg21.link/P1091

3. If the initialiser's type contains only non-static data members:

```
struct Point  {
    double x;
    double y;
};

Point GetStartPoint() {
    return { 0.0, 0.0 };
}

const auto [x, y] = GetStartPoint();
```

x and y refer to `Point::x` and `Point::y` from the `Point` structure.

The class doesn't have to be `POD`, but the number of identifiers must equal to the number of non-static data members. The members must also be accessible from the given context.

Note: In C++17, initially, you could use structured bindings to bind to class members as long as they were public. That could be a problem when you wanted to access such members in a context of friend functions, or even inside a struct implementation. This issue was recognised quickly as a defect, and it's now fixed in C++17. See P0969R0[2].

Examples

This section will show you a few examples where structured bindings are helpful. In the first one, we'll use them to write more expressive code, and in the next one, you'll see how to provide API for your class to support structured bindings.

Expressive Code With Structured Bindings

If you have a map of elements, you might know that internally they are stored as pairs of `<const Key, ValueType>`.

[2] https://wg21.link/P0969R0

Now, when you iterate through elements of that map:

```
for (const auto& elem : myMap) { ... }
```

You need to write `elem.first` and `elem.second` to refer to the key and value. One of the **coolest use cases** of structured binding is that we can use it inside a range based for loop:

```
std::map<KeyType, ValueType> myMap;
// C++14:
for (const auto& elem : myMap) {
    // elem.first - is velu key
    // elem.second - is the value
}
// C++17:
for (const auto& [key,val] : myMap) {
    // use key/value directly
}
```

In the above example, we bind to a pair of `[key, val]` so we can use those names in the loop. Before C++17 you had to operate on an iterator from the map - which returns a pair `<first, second>`. Using the real names `key/value` is more expressive.

The above technique can be used in:

Chapter General Language Features/city_map_iterate.cpp

```
#include <map>
#include <iostream>
#include <string>

int main() {
    const std::map<std::string, int> mapCityPopulation {
        { "Beijing", 21'707'000 },
        { "London", 8'787'892 },
        { "New York", 8'622'698 }
    };

    for (auto&[city, population] : mapCityPopulation)
        std::cout << city << ": " << population << '\n';
}
```

In the loop body, you can safely use `city` and `population` variables.

Providing Structured Binding Interface for Custom Class

As mentioned earlier, you can provide Structured Binding support for a custom class.

To do that you have to define `get<N>`, `std::tuple_size` and `std::tuple_element` specialisations for your type.

For example, if you have a class with three members, but you'd like to expose only its public interface:

Chapter Chapter General Language Features/custom_structured_bindings.cpp

```
class UserEntry {
public:
    void Load() { }

    std::string GetName() const { return name; }
    unsigned GetAge() const { return age; }
private:
    std::string name;
    unsigned age { 0 };
    size_t cacheEntry { 0 }; // not exposed
};
```

The interface for Structured Bindings:

Chapter Chapter General Language Features/custom_structured_bindings.cpp

```
// with if constexpr:
template <size_t I> auto get(const UserEntry& u) {
    if constexpr (I == 0) return u.GetName();
    else if constexpr (I == 1) return u.GetAge();
}

namespace std {
    template <> struct tuple_size<UserEntry> : integral_constant<size_t, 2> { };

    template <> struct tuple_element<0,UserEntry> { using type = std::string; };
    template <> struct tuple_element<1,UserEntry> { using type = unsigned; };
}
```

`tuple_size` specifies how many fields are available, `tuple_element` defines the type for a specific element and `get<N>` returns the values.

Alternatively, you can also use explicit `get<>` specialisations rather than `if constexpr`:

```
template<> string get<0>(const UserEntry &u)  { return u.GetName(); }
template<> unsigned get<1>(const UserEntry &u) { return u.GetAge(); }
```

For a lot of types, writing two (or several) functions might be more straightforward than using `if constexpr`.

Now you can use `UserEntry` in a structured binding, for example:

```
UserEntry u;
u.Load();
auto [name, age] = u; // read access
std:: cout << name << ", " << age << '\n';
```

This example only allows read access of the class. If you want write access, then the class should also provide accessors that return references to members. Later you have to implement `get` with references support.

The code in this section used `if constexpr`, you can read more about this powerful feature in the next chapter: Templates: `if constexpr`.

Extra Info

The change was proposed in: P0217[3](wording), P0144[4](reasoning and examples), P0615[5](renaming "decomposition declaration" with "structured binding declaration").

[3] https://wg21.link/p0217
[4] https://wg21.link/p0144
[5] https://wg21.link/p0615

Init Statement for `if` and `switch`

C++17 provides new versions of the if and switch statements:

```
if (init; condition)
```

And

```
switch (init; condition)
```

In the `init` section you can specify a new variable, similarly to the init section in for loop. Then check the variable in the `condition` section. The variable is visible only in `if`/`else` scope.

To achieve a similar result, before C++17, you had to write:

```
{
    auto val = GetValue();
    if (condition(val))
        // on success
    else
        // on false...
}
```

Please notice that `val` has a separate scope, without that it 'leaks' to enclosing scope.

Now, in C++17, you can write:

```
if (auto val = GetValue(); condition(val))
    // on success
else
    // on false...
```

Now, `val` is visible only inside the `if` and `else` statements, so it doesn't 'leak.' `condition` might be any boolean condition.

Why is this useful?

Let's say you want to search for a few things in a string:

```
const std::string myString = "My Hello World Wow";

const auto pos = myString.find("Hello");
if (pos != std::string::npos)
    std::cout << pos << " Hello\n"

const auto pos2 = myString.find("World");
if (pos2 != std::string::npos)
    std::cout << pos2 << " World\n"
```

You have to use different names for `pos` or enclose it with a separate scope:

```
{
    const auto pos = myString.find("Hello");
    if (pos != std::string::npos)
        std::cout << pos << " Hello\n"
}

{
    const auto pos = myString.find("World");
    if (pos != std::string::npos)
        std::cout << pos << " World\n"
}
```

The new if statement will make that additional scope in one line:

```
if (const auto pos = myString.find("Hello"); pos != std::string::npos)
    std::cout << pos << " Hello\n";

if (const auto pos = myString.find("World"); pos != std::string::npos)
    std::cout << pos << " World\n";
```

As mentioned before, the variable defined in the if statement is also visible in the `else` block. So you can write:

```
if (const auto pos = myString.find("World"); pos != std::string::npos)
    std::cout << pos << " World\n";
else
    std::cout << pos << " not found!!\n";
```

Plus, you can use it with structured bindings (following Herb Sutter code[6]):

```
// better together: structured bindings + if initializer
if (auto [iter, succeeded] = mymap.insert(value); succeeded) {
    use(iter);  // ok
    // ...
} // iter and succeeded are destroyed here
```

In the above example, you can refer to `iter` and `succeeded` rather than `pair.first` and `pair.second` that is returned from `mymap.insert`.

As you can see, structured bindings and tuples allow you to create even more variables in the init section of the if-statement. But is the code easier to read that way?

For example:

```
string str = "Hi World";
if (auto [pos, size] = pair(str.find("Hi"), str.size()); pos != string::npos)
    std::cout << pos << " Hello, size is " << size;
```

We can argue that putting more code into the init section makes the code less readable, so pay attention to such cases.

Extra Info

The change was proposed in: P0305R1[7].

[6] https://herbsutter.com/2016/06/30/trip-report-summer-iso-c-standards-meeting-oulu/
[7] http://wg21.link/p0305r1

Inline Variables

With Non-Static Data Member Initialisation introduced in C++11, you can now declare and initialise member variables in one place:

```
class User {
    int _age {0};
    std::string _name {"unknown"};
};
```

However, with static variables (or `const static`) you need a declaration and then a definition in the implementation file.

C++11 with `constexpr` keyword allows you to declare and define static variables in one place, but it's limited to constant expressions only.

Previously, only methods/functions could be specified as `inline`, but now you can do the same with variables, inside a header file.

From the proposal P0386R2[8]:

> A variable declared inline has the same semantics as a function declared inline: it can be defined, identically, in multiple translation units, must be defined in every translation unit in which it is used, and the behaviour of the program is as if there was exactly one variable.

For example:

```
// inside a header file:
struct MyClass {
    static const int sValue;
};

// later in the same header file:
inline int const MyClass::sValue = 777;
```

Or even declaration and definition in one place:

[8] http://wg21.link/p0386r2

```
struct MyClass {
    inline static const int sValue = 777;
};
```

Also, note that `constexpr` variables are `inline` implicitly, so there's no need to use `constexpr inline myVar = 10;`.

An `inline` variable is also more flexible than a `constexpr` variable as it doesn't have to be initialised with a constant expression. For example, you can initialise an `inline` variable with `rand()`, but it's not possible to do the same with `constexpr` variable.

How Can it Simplify the Code?

A lot of header-only libraries can limit the number of hacks (like using inline functions or templates) and switch to using inline variables.

For example:

```
class MyClass {
    static inline int Seed(); // static method
};

inline int MyClass::Seed() {
    static const int seed = rand();
    return seed;
}
```

Can be changed into:

```
class MyClass {
    static inline int seed = rand();
};
```

C++17 guarantees that `MyClass::seed` will have the same value (generated at runtime) across all the compilation units!

Extra Info

The change was proposed in: P0386R2[9].

[9] http://wg21.link/p0386r2

`constexpr` Lambda Expressions

Lambda expressions were introduced in C++11, and since that moment they've become an essential part of modern C++. Another significant feature of C++11 is the `constexpr` specifier, which is used to express that a function or value can be computed at compile-time. In C++17, the two elements are allowed to exist together, so your lambda can be invoked in a constant expression context.

In C++11/14 the following code doesn't compile, but works with C++17:

Chapter General Language Features/lambda_square.cpp

```
int main () {
    constexpr auto SquareLambda = [] (int n) { return n*n; };
    static_assert(SquareLambda(3) == 9, "");
}
```

Since C++17 lambda expressions (their call operator `operator()`) that follow the rules of standard `constexpr` functions are implicitly declared as `constexpr`.

What are the limitations of `constexpr` functions? Here's a summary (from 10.1.5 The constexpr specifier [dcl.constexpr][10]):

- they cannot be virtual
- their return type shall be a literal type
- their parameter types shall be a literal type
- their function bodies cannot contain: `asm` definition, a `goto` statement, try-block, or a variable that is a non-literal type or static or thread storage duration

In practice, in C++17, if you want your function or lambda to be executed at compile-time, then the body of this function shouldn't invoke any code that is not `constexpr`. For example, you cannot allocate memory dynamically or throw exceptions.

`constexpr` lambda expressions are also covered in the Other Changes Chapter and in a free ebook: C++ Lambda Story[11].

Extra Info

The change was proposed in: P0170[12].

[10] https://timsong-cpp.github.io/cppwp/n4659/dcl.constexpr#3
[11] https://leanpub.com/cpplambda
[12] http://wg21.link/p0170

Capturing `[*this]` in Lambda Expressions

When you write a lambda inside a class method, you can reference a member variable by capturing `this`. For example:

Chapter General Language Features/capture_this.cpp

```
struct Test  {
    void foo() {
        std::cout << m_str  << '\n';
        auto addWordLambda = [this]() { m_str += "World"; };
        addWordLambda ();
        std::cout << m_str  << '\n';
    }

    std::string m_str {"Hello "};
};
```

In the line with `auto addWordLambda = [this]() {... }` we capture `this` pointer and later we can access `m_str`.

Please notice that we captured `this` by value... to a pointer. You have access to the member variable, not its copy. The same effect happens when you capture by `[=]` or `[&]`. That's why when you call `foo()` on some `Test` object then you'll see the following output:

```
Hello
Hello World
```

`foo()` prints `m_str` two times. The first time we see `"Hello"`, but the next time it's `"Hello World"` because the lambda `addWordLambda` changed it.

How about more complicated cases? Do you know what will happen with the following code?

Returning a Lambda From a Method

```
#include <iostream>

struct Baz {
    auto foo() {
        return [=] { std::cout << s << '\n'; };
    }
    std::string s;
};

int main() {
   auto f1 = Baz{"ala"}.foo();
   f1();
}
```

The code declares a `Baz` object and then invokes `foo()`. Please note that `foo()` returns a lambda that captures a member of the class.

A capturing block like `[=]` suggests that we capture variables by value, but if you access members of a class in a lambda expression, then it does this implicitly via the `this` pointer. So we captured a copy of `this` pointer, which is a dangling pointer as soon as we exceed the lifetime of the `Baz` object.

In C++17 you can write: `[*this]` and that will capture **a copy** of the whole object.

```
auto lam = [*this]() { std::cout << s; };
```

In C++14, the only way to make the code safer is init capture `*this`:

```
auto lam = [self=*this] { std::cout << self.s; };
```

Capturing `this` might get tricky when a lambda can outlive the object itself. This might happen when you use async calls or multithreading.

In C++20 (see P0806[13]) you'll also see an extra warning if you capture `[=]` in a method. Such expression captures the `this` pointer, and it might not be exactly what you want.

Extra Info

The change was proposed in: P0018[14].

[13] https://wg21.link/P0806
[14] http://wg21.link/p0018

Nested Namespaces

Namespaces allow grouping types and functions into separate logical units.

For example, it's common to see that each type or function from a library XY will be stored in a namespace `xy`. Like in the below case, where there's `SuperCompressionLib` and it exposes functions called `Compress()` and `Decompress()`:

```
namespace SuperCompressionLib {
    bool Compress();
    bool Decompress();
}
```

Things get interesting if you have two or more nested namespaces.

```
namespace MySuperCompany {
    namespace SecretProject {
        namespace SafetySystem {
            class SuperArmor {
                // ...
            };
            class SuperShield {
                // ...
            };
        } // SafetySystem
    } // SecretProject
} // MySuperCompany
```

With C++17 nested namespaces can be written more compactly:

```
namespace MySuperCompany::SecretProject::SafetySystem {
    class SuperArmor {
    // ...
    };
    class SuperShield {
    // ...
    };
}
```

Such syntax is comfortable, and it will be easier to use for developers that have experience in languages like C# or Java.

In C++17 also the Standard Library was "compacted" in several places by using the new nested namespace feature:

For example, for `regex`.

In C++17 it's defined as:

```
namespace std::regex_constants {
    typedef T1 syntax_option_type;
    // ...
}
```

Before C++17 the same was declared as:

```
namespace std {
    namespace regex_constants {
        typedef T1 syntax_option_type;
        // ...
    }
}
```

The above nested declarations appear in the C++ Specification, but it might look different in an STL implementation.

Extra Info

The change was proposed in: N4230[15].

[15] http://wg21.link/N4230

`__has_include` Preprocessor Expression

If your code has to work under two different compilers, then you might experience two different sets of available features and platform-specific changes.

In C++17 you can use `__has_include` preprocessor constant expression to check if a given header exists:

```
#if __has_include(<header_name>)
#if __has_include("header_name")
```

`__has_include` was available in Clang as an extension for many years, but now it was added to the Standard. It's a part of "feature testing" helpers that allows you to check if a particular C++ feature or a header is available. If a compiler supports this macro, then it's accessible even without the C++17 flag, that's why you can check for a feature also if you work in C++11, or C++14 "mode".

As an example, we can test if a platform has `<charconv>` header that declares C++17's low-level conversion routines:

Chapter General Language Features/has_include.cpp

```
#if defined __has_include
#    if __has_include(<charconv>)
#        define has_charconv 1
#        include <charconv>
#    endif
#endif

std::optional<int> ConvertToInt(const std::string& str) {
    int value { };
    #ifdef has_charconv
        const auto last = str.data() + str.size();
        const auto res = std::from_chars(str.data(), last, value);
        if (res.ec == std::errc{} && res.ptr == last)
            return value;
    #else
        // alternative implementation...
    #endif

    return std::nullopt;
}
```

In the above code, we declare `has_charconv` based on the `__has_include` condition. If the header is not there, we need to provide an alternative implementation for `ConvertToInt`. You can check this code against GCC 7.1 and GCC 9.1 and see the effect as GCC 7.1 doesn't expose the `charconv` header.

Note: In the above code we cannot write:

```
#if defined __has_include && __has_include(<charconv>)
```

As in older compilers - that don't support `__has_include` we'd get a compile error. The compiler will complain that since `__has_include` is not defined and the whole expression is wrong.

Another important thing to remember is that sometimes a compiler might provide a header stub. For example, in C++14 mode the `<execution>` header might be present (it defines C++17 parallel algorithm execution modes), but the whole file will be empty (due to `ifdefs`). If you check for that file with `__has_include` and use C++14 mode, then you'll get a wrong result.

In C++20 we'll have standardised feature test macros that simplify checking for various C++ parts. For example, to test for `std::any` you can use `__cpp_lib_any`, for lambda support there's `__cpp_lambdas`. There's even a macro that checks for attribute support: `__has_cpp_attribute( attrib-name)`. GCC, Clang and Visual Studio exposes many of the macros already, even before C++20 is ready. Read more in Feature testing (C++20) - cppreference[16]

`__has_include` along with feature testing macros might greatly simplify multiplatform code that usually needs to check for available platform elements.

Extra Info

`__has_include` was proposed in: P0061[17].

[16] https://en.cppreference.com/w/cpp/feature_test
[17] http://wg21.link/p0061

Compiler support

Feature	GCC	Clang	MSVC	C++ Builder
Structured Binding Declarations	7.0	4.0	VS 2017 15.3	10.3 Rio
Init-statement for if/switch	7.0	3.9	VS 2017 15.3	10.3 Rio
Inline variables	7.0	3.9	VS 2017 15.5	10.3 Rio
`constexpr` Lambda Expressions	7.0	5.0	VS 2017 15.3	10.3 Rio
Lambda Capture of `*this`	7.0	3.9	VS 2017 15.3	10.3 Rio
Nested namespaces	6.0	3.6	VS 2015	10.3 Rio
`has_include`	5	Yes	VS 2017 15.3	10.3 Rio

5. Templates

Do you work with templates and/or meta-programming?

If your answer is “YES,” then you might be pleased with the updates from C++17.

The new standard introduces many enhancements that make template programming much easier and more expressive.

In this chapter, you’ll learn:

- Template argument deduction for class templates
- `template<auto>`
- Fold expressions
- `if constexpr` - the compile-time if for C++!
- Plus some smaller, detailed improvements and fixes

Template Argument Deduction for Class Templates

C++17 filled a gap in the deduction rules for templates. Now, the template argument deduction can occur for class templates and not just for functions. That also means that a lot of your code that uses `make_Type` functions can now be removed.

For instance, to create an `std::pair` object, it was usually more comfortable to write:

```
auto myPair = std::make_pair(42, "hello world"s);
```

Rather than:

```
std::pair<int, std::string> myPair{42, "hello world"};
```

Because `std::make_pair()` is a template function, the compiler can perform the deduction of function template arguments and there's no need to write:

```
auto myPair = std::make_pair<int, std::string>(42, "hello world");
```

Now, since C++17, the conformant compiler will nicely deduce the template parameter types for class templates too!

The feature is called *"Class Template Argument Deduction"* or *CTAD* in short.

In our example, you can now write:

```
using namespace std::string_literals;
std::pair myPair{42, "hello world"s};     // deduced automatically!
```

CTAD also works with copy initialisation and when allocating memory through `new()`:

```
auto otherPair = std::pair{42, "Hello"s}; // also deduced
auto ptr = new std::pair{42, "World"s};   // for new
```

CTAD can substantially reduce complex constructions like:

```
// lock guard:
std::shared_timed_mutex mut;
std::lock_guard<std::shared_timed_mutex> lck(mut);

// array:
std::array<int, 3> arr {1, 2, 3};
```

Can now become:

```
std::shared_timed_mutex mut;
std::lock_guard lck(mut);

std::array arr { 1, 2, 3 };
```

Note, that partial deduction cannot happen, you have to specify all the template parameters or none:

```
std::tuple t(1, 2, 3);              // OK: deduction
std::tuple<int,int,int> t(1, 2, 3); // OK: all arguments are provided
std::tuple<int> t(1, 2, 3);         // Error: partial deduction
```

With this feature, a lot of `make_Type` functions might not be needed - especially those that "emulate" template deduction for classes.

Still, there are factory functions that do additional work. For example, `std::make_shared` - it not only creates `shared_ptr`, but also makes sure the control block, and the pointed object are allocated in one memory region:

```
// control block and int might be in different places in memory
std::shared_ptr<int> p(new int{10});

// the control block and int are in the same contiguous memory section
auto p2 = std::make_shared<int>(10);
```

How does template argument deduction for classes work?

Let's enter the "Deduction Guides" area.

Deduction Guides

The compiler uses special rules called "*Deduction Guides*" to work out parameter types.

There are two types of guides: compiler-generated (implicitly generated) and user-defined.

To understand how the compiler uses the guides, let's look at a simplified deduction guide[1] for `std::array`:

```
template<typename T, typename... U>
array(T, U...) ->
    array<enable_if_t<(is_same_v<T, U> && ...), T>, 1 + sizeof...(U)>;
```

The syntax looks like a template function with a trailing return type. The compiler treats such "imaginary" function as a candidate for the parameters. If the pattern matches, then the found types are returned from the deduction.

In our case when you write:

```
std::array arr {1, 2, 3, 4};
```

Then, assuming `T` and `U...` arguments are of the same type, we can build up an array object of the type `std::array<int, 4>`.

In most cases, you can rely on the compiler to generate automatic deduction guides. They will be created for each constructor (also copy/move) of the primary class template. Please note that classes that are specialised or partially specialised won't work here.

As mentioned, you might also write custom deduction guides. A classic example is a deduction of `std::string` rather than `const char*`:

```
template<typename T> struct MyType {
  T str;
};

// custom deduction guide
MyType(const char *) -> MyType<std::string>;
MyType t{"Hello World"}; // deduces std::string
```

Without the custom deduction `T` would be deduced as `const char*`.

[1] simplified version of libstdc++ code https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/std/array

Another example of custom deduction guide comes from the `overload` pattern[2]:

```
template<class... Ts>
struct overload : Ts... { using Ts::operator()...; };

template<class... Ts>
overload(Ts...) -> overload<Ts...>; // deduction guide
```

The `overload` class inherits from other classes `Ts...` and then exposes their `operator()`. The custom deduction guide is used here to "transform" a list of callable types into the list of classes that we can derive from.

CTAD Limitations

In C++17 template argument deduction for classes has the following limitations:

- it doesn't work with template aggregate types
- the deduction doesn't include inheriting constructors
- it doesn't work with template aliases

Those limitations will be removed in C++20 through the already accepted proposal: P1021[3].

Extra Info

The CTAD feature was proposed in: P0091R3[4] and P0433 - Deduction Guides in the Standard Library[5].

Please note that while a compiler might declare full support for Template Argument Deduction for Class Templates, its corresponding STL implementation might still lack of custom deduction guides for some STL types. See the Compiler Support section at the end of the chapter.

[2] Read more about this pattern in the chapter about `std::variant`, section related to `std::visit()`
[3] https://wg21.link/P1021
[4] http://wg21.link/p0091r3
[5] https://wg21.link/p0433

Fold Expressions

C++11 introduced variadic templates, which is a powerful feature, especially if you want to work with a variable number of input template parameters to a function. For example, previously (pre C++11) you had to write several different versions of a template function (one for one parameter, another for two parameters, another for three params…).

Still, variadic templates required some additional code when you wanted to implement "recursive" functions like `sum`, `all`. You had to specify rules for the recursion.

For example:

```
auto SumCpp11() {
    return 0;
}

template<typename T1, typename... T>
auto SumCpp11(T1 s, T... ts) {
    return s + SumCpp11(ts...);
}
```

And with C++17 we can write much simpler code:

```
template<typename ...Args> auto sum(Args ...args) {
    return (args + ... + 0);
}

// or even:
template<typename ...Args> auto sum2(Args ...args) {
    return (args + ...);
}
```

The following variations of fold expressions[6] with binary operators (`op`) exist:

Expression	Name	Expansion
`(... op e)`	unary left fold	`((e1 op e2) op ...) op eN`
`(init op ... op e)`	binary left fold	`(((init op e1) op e2) op ...) op eN`
`(e op ...)`	unary right fold	`e1 op (... op (eN-1 op eN))`
`(e op ... op init)`	binary right fold	`e1 op (... op (eN-1 op (eN op init)))`

[6] https://en.cppreference.com/w/cpp/language/fold

`op` is any of the following 32 binary operators: `+ - * / % ^ & | = < > << >> += -= *= /= %= ^= &= |= <<= >>= == != <= >= && || , .* ->*`. In a binary fold, both ops must be the same.

For example, when you write:

```
template<typename ...Args> auto sum2(Args ...args) {
    return (args + ...); // unary right fold over '+'
}

auto value = sum2(1, 2, 3, 4);
```

The template function is expanded into:

```
auto value = 1 + (2 + (3 + 4));
```

Also by default we get the following values for empty parameter packs:

Operator	default value
&&	`true`
||	`false`
,	`void()`
any other	ill-formed code

That's why you cannot call `sum2()` without any parameters, as the unary fold over operator `+` doesn't have any default value for the empty parameter list.

More Examples

Here's a quite nice implementation of a `printf` using folds P0036R0[7]:

[7] http://wg21.link/p0036r0

```
template<typename ...Args>
void FoldPrint(Args&&... args) {
    (std::cout << ... << std::forward<Args>(args)) << '\n';
}

FoldPrint("hello", 10, 20, 30);
```

However, the above `FoldPrint` will print arguments one by one, without any separator. For the above function call, you'll see `"hello102030"` on the output.

If you want separators and more formatting options, you have to alter the printing technique and use fold over comma:

```
template<typename ...Args>
void FoldSeparateLine(Args&&... args) {
    auto separateLine = [](const auto& v) {
        std::cout << v << '\n';
    };
    (... , separateLine (std::forward<Args>(args))); // over comma operator
}
```

The technique with fold over the comma operator is handy. Another example of it might be a special version of `push_back`:

```
template<typename T, typename... Args>
void push_back_vec(std::vector<T>& v, Args&&... args) {
    (v.push_back(std::forward<Args>(args)), ...);
}

std::vector<float> vf;
push_back_vec(vf, 10.5f, 0.7f, 1.1f, 0.89f);
```

In general, fold expression allows you to write cleaner, shorter and probably more comfortable to read the code.

Extra Info

The change was proposed in: N4295[8] and P0036R0[9].

[8] https://wg21.link/n4295
[9] https://wg21.link/p0036r0

`if constexpr`

This is a big one!

The compile-time if for C++!

The feature allows you to discard branches of an if statement at compile-time based on a constant expression condition.

```
if constexpr (cond)
     statement1; // Discarded if cond is false
else
     statement2; // Discarded if cond is true
```

For example:

```
template <typename T>
auto get_value(T t) {
    if constexpr (std::is_pointer_v<T>)
        return *t;
    else
        return t;
}
```

`if constexpr` has the potential to simplify a lot of template code - especially when tag dispatching, SFINAE or preprocessor techniques are used.

Why Compile Time If?

At first, you may ask, why do we need `if constexpr` and those complex templated expressions... wouldn't a regular `if` work?

Here's a code example:

```
template <typename Concrete, typename... Ts>
std::unique_ptr<Concrete> constructArgs(Ts&&... params) {
    if (std::is_constructible_v<Concrete, Ts...>) // regular `if`
        return std::make_unique<Concrete>(std::forward<Ts>(params)...);
    else
        return nullptr;
}
```

The above routine is an "updated" version of `std::make_unique`: it returns `std::unique_ptr` when the parameters allow it to construct the wrapped objects, or it returns `nullptr`.

Below there's simple code that tests `constructArgs`:

```
class Test {
public:
    Test(int, int) { }
};

int main() {
    auto p = constructArgs<Test>(10, 10, 10); // 3 args!
}
```

The code tries to build `Test` out of three parameters, but please notice that `Test` has only constructor that takes two `int` arguments.

When compiling you might get a similar compiler error:

```
In instantiation of 'typename std::_MakeUniq<_Tp>::__single_object
std::make_unique(_Args&& ...) [with _Tp = Test; _Args = {int, int, int};
typename std::_MakeUniq<_Tp>::__single_object
 = std::unique_ptr<Test, std::default_delete<Test> >]':

main.cpp:8:40: required from 'std::unique_ptr<_Tp>
    constructArgs(Ts&& ...) [with Concrete = Test; Ts = {int, int, int}]'
```

Let's try to understand this error message. After the template deduction the compiler compiles the following code:

```
if (std::is_constructible_v<Concrete, int, int, int>)
   return std::make_unique<Concrete>(10, 10, 10);
else
   return nullptr;
```

During the runtime, the `if` branch won't be executed - as `is_constructible_v` returns `false`, yet the code in the branch must compile.

That's why we need `if constexpr`, to "discard" code and compile only the matching statement.

To fix the code you have to add `constexpr`:

```
template <typename Concrete, typename... Ts>
std::unique_ptr<Concrete> constructArgs(Ts&&... params) {
  if constexpr (std::is_constructible_v<Concrete, Ts...>) // fixed!
      return std::make_unique<Concrete>(std::forward<Ts>(params)...);
   else
       return nullptr;
}
```

Now, the compiler evaluates the `if constexpr` condition at compile-time and for the expression `auto p = constructArgs<Test>(10, 10, 10);` the whole `if` branch will be "removed" from the second step of the compilation process.

To be precise, the code in the discarded branch is not entirely removed from the compilation phase. Only expressions that are dependent on the template parameter used in the condition are not instantiated. The syntax must always be valid.

For example:

```
template <typename T>
void Calculate(T t) {
    if constexpr (std::is_integral_v<T>) {
        // ...
        static_assert(sizeof(int) == 100);
    }
    else {
        execute(t);
        strange syntax
    }
}
```

In the above artificial code, if the type `T` is `int`, then the `else` branch is discarded, which means `execute(t)` won't be instantiated. But the line `strange syntax` will still be compiled (as it's not dependent on `T`) and that's why you'll get a compile error about that.

Furthermore, another compilation error will come from `static_assert`, the expression is also not dependent on `T`, and that's why it will always be evaluated.

Template Code Simplification

Before C++17 if you had several versions of an algorithm - depending on the type requirements - you could use SFINAE or tag dispatching to generate a dedicated overload resolution set.

For example:

Chapter Templates/sfinae_example.cpp

```
template <typename T>
std::enable_if_t<std::is_integral_v<T>, T> simpleTypeInfo(T t) {
    std::cout << "foo<integral T> " << t << '\n';
    return t;
}

template <typename T>
std::enable_if_t<!std::is_integral_v<T>, T> simpleTypeInfo(T t) {
    std::cout << "not integral \n";
    return t;
}
```

In the above example, we have two function implementations, but only one of them will end up in the overload resolution set. If `std::is_integral_v` is true for the `T` type, then the top function is taken, and the second one rejected due to SFINAE.

The same thing can happen when using tag dispatching:

Chapter Templates/tag_dispatching_example.cpp

```
template <typename T>
T simpleTypeInfoTagImpl(T t, std::true_type) {
    std::cout << "foo<integral T> " << t << '\n';
    return t;
}

template <typename T>
T simpleTypeInfoTagImpl(T t, std::false_type) {
    std::cout << "not integral \n";
    return t;
}

template <typename T>
T simpleTypeInfoTag(T t) {
    return simpleTypeInfoTagImpl(t, std::is_integral<T>{});
}
```

Now, instead of SFINAE, we generate a unique type tag for the condition: `true_type` or `false_type`. Depending on the result, only one implementation is selected.

We can now simplify this pattern with `if constexpr`:

```
template <typename T>
T simpleTypeInfo(T t) {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "foo<integral T> " << t << '\n';
    }
    else {
        std::cout << "not integral \n";
    }
    return t;
}
```

Writing template code becomes more “natural” and doesn’t require that many “tricks”.

Examples

Let’s see a couple of examples:

Line Printer

You might have already seen the below example in the Jump Start section at the beginning of this Part of the book. Let's dive into the details and see how the code works.

```
template<typename T> void linePrinter(const T& x) {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "num: " << x << '\n';
    }
    else if constexpr (std::is_floating_point_v<T>) {
        const auto frac = x - static_cast<long>(x);
        std::cout << "flt: " << x << ", frac " << frac << '\n';
    }
    else if constexpr(std::is_pointer_v<T>) {
        std::cout << "ptr, ";
        linePrinter(*x);
    }
    else {
        std::cout << x << '\n';
    }
}
```

`linePrinter` uses `if constexpr` to check the input type. Based on that, we can output additional messages. An interesting thing happens with the pointer type - when a pointer is detected the code dereferences it and then calls `linePrinter` recursively.

Declaring Custom `get<N>` Functions

The structured binding expression works for simple structures that have all public members, like

```
struct S {
    int n;
    std::string s;
    float d;
};

S s;
auto [a, b, c] = s;
```

However, if you have a custom type (with private members), then it's also possible to override `get<N>` functions so that structured binding can work. Here's some code to demonstrate this idea:

```
class MyClass {
public:
    int GetA() const { return a; }
    float GetB() const { return b; }

private:
    int a;
    float b;
};

template <std::size_t I> auto get(MyClass& c) {
    if constexpr (I == 0)      return c.GetA();
    else if constexpr (I == 1) return c.GetB();
}

// specialisations to support tuple-like interface
namespace std {
    template <> struct tuple_size<MyClass> : integral_constant<size_t, 2> { };

    template <> struct tuple_element<0,MyClass> { using type = int; };
    template <> struct tuple_element<1,MyClass> { using type = float; };
}
```

In the above code you have the advantage of having everything in one function. It's also possible to do it as template specialisations:

```
template <> auto& get<0>(MyClass &c) { return c.GetA(); }
template <> auto& get<1>(MyClass &c) { return c.GetB(); }
```

For more examples you can read the chapter about Replacing `std::enable_if` with `if constexpr` and also the chapter Structured Bindings - the section about custom `get<N>` specialisations.

You can also see the following article: Simplify code with if constexpr in C++17[10]

Extra Info

The change was proposed in: P0292R2[11].

[10] https://www.bfilipek.com/2018/03/ifconstexpr.html
[11] https://wg21.link/p0292r2

Declaring Non-Type Template Parameters With `auto`

This is another part of the strategy to use `auto` everywhere. With C++11 and C++14, you can use it to deduce variables or even return types automatically, plus there are also generic lambdas. Now you can also use it for deducing non-type template parameters.

For example:

```
template <auto value> void f() { }
f<10>();      // deduces int
```

This is useful, as you don't have to have a separate parameter for the type of non-type parameter. Like in C++11/14:

```
template <typename Type, Type value> constexpr Type TConstant = value;
constexpr auto const MySuperConst = TConstant<int, 100>;
```

With C++17 it's a bit simpler:

```
template <auto value> constexpr auto TConstant = value;
constexpr auto const MySuperConst = TConstant<100>;
```

There's no need to write `Type` explicitly.

As one of the advanced uses a lot of papers, and articles point to an example of heterogeneous compile time list:

```
template <auto ... vs> struct HeterogenousValueList {};
using MyList = HeterogenousValueList<'a', 100, 'b'>;
```

Before C++17 it was not possible to declare such list directly, some wrapper class would have had to be provided first.

Extra Info

The change was proposed in: P0127R2[12]. In P0127R1[13], you can find some more examples and reasoning.

[12] https://wg21.link/p0127r2
[13] https://wg21.link/p0127r1

Other Changes

In C++17 there are also other language and library features related to templates that are worth mentioning:

Allow `typename` in a Template Template Parameters.

Allows you to use `typename` instead of `class` when declaring a template template parameter. Normal type parameters can use them interchangeably, but template template parameters were restricted to `class`.

More information in N4051[14].

Allow Constant Evaluation for all Non-Type Template Arguments

Remove syntactic restrictions for pointers, references, and pointers to members that appear as non-type template parameters.

More information in N4268[15].

Variable Templates for Traits

All the type traits that yields `::value` got accompanying `_v` variable templates. For example:

`std::is_integral<T>::value` has `std::is_integral_v<T>`

`std::is_class<T>::value` has `std::is_class_v<T>`

This improvement already follows the `_t` suffix additions in C++14 (template aliases) to type traits that returns `::type`. Such change can considerably shorten template code.

More information in P0006R0[16].

[14] https://wg21.link/n4051
[15] https://wg21.link/n4268
[16] https://wg21.link/p0006r0

Pack Expansions in Using Declarations

The feature is an enhancement for variadic templates and parameter packs.

The compiler will now support the `using` keyword in pack expansions:

```
template<class... Ts> struct overloaded : Ts... {
    using Ts::operator()...;
};
```

The `overloaded` class exposes all overloads for `operator()` from the base classes. Before C++17, you would have to use recursion for parameter packs to achieve the same result. The `overloaded` pattern is a very useful enhancement for `std::visit`, read more in the "Overload" section in the Variant chapter.

More information in P0195[17].

Logical Operation Metafunctions

C++17 adds handy template metafunctions:

- `template<class... B> struct conjunction;` - logical `AND`
- `template<class... B> struct disjunction;` - logical `OR`
- `template<class B> struct negation;` - logical negation

Here's an example, based on the code from the proposal:

```
template<typename... Ts>
std::enable_if_t<std::conjunction_v<std::is_same<int, Ts>...> >
PrintIntegers(Ts ... args) {
    (std::cout << ... << args) << '\n';
}
```

The above function `PrintIntegers` works with a variable number of arguments, but they all have to be of type `int`.

The helper metafunctions can increase the readability of advanced template code. They are available in `<type_traits>` header.

More information in P0013[18].

[17] https://wg21.link/P0195
[18] https://wg21.link/P0013

`std::void_t` Transformation Trait

A surprisingly simple[19] metafunction that maps a list of types into `void`:

```
template< class... >
using void_t = void;
```

`void_t` is very handy to SFINAE ill-formed types. For example it might be used to detect a function overload:

```
void Compute(int &) { } // example function

template <typename T, typename = void>
struct is_compute_available : std::false_type {};

template <typename T>
struct is_compute_available<T,
           std::void_t<decltype(Compute(std::declval<T>())) >>
               : std::true_type {};

static_assert(is_compute_available<int&>::value);
static_assert(!is_compute_available<double&>::value);
```

`is_compute_available` checks if a `Compute()` overload is available for the given template parameter. If the expression `decltype(Compute(std::declval<T>()))` is valid, then the compiler will select the template specialisation. Otherwise, it's SFINEed, and the primary template is chosen.

More information in N3911[20].

[19] Compilers that don't implement a fix for CWG 1558 (for C++14) might need a more complicated version of it.
[20] https://wg21.link/n3911

Compiler Support

Feature	GCC	Clang	MSVC	C++ Builder
Template argument deduction for class templates	7.0/8.0[21]	5.0	VS 2017 15.7	10.3 Rio
Deduction Guides in the Standard Library	8.0[22]	7.0/in progress[23]	VS 2017 15.7	10.3 Rio
Declaring non-type template parameters with auto	7.0	4.0	VS 2017 15.7	10.3 Rio
Fold expressions	6.0	3.9	VS 2017 15.5	10.3 Rio
`if constexpr`	7.0	3.9	VS 2017	10.3 Rio

[21] Additional improvements for Template Argument Deduction for Class Templates happened in GCC 8.0, P0512R0.

[22] Deduction Guides are not listed in the status pages of LibSTDC++, so we can assume they were implemented as part of Template argument deduction for class templates.

[23] The status page for LibC++ mentions that `<string>`, sequence containers, container adaptors and `<regex>` portions have been implemented so far.

6. Standard Attributes

Code annotations - attributes - are probably not the best-known feature of C++. However, they might be handy for expressing additional information for the compiler and also for other programmers. Since C++11, there has been a standard way of specifying attributes. And in C++17 we got even more useful additions.

In this chapter, you'll learn:

- What are the attributes in C++
- Vendor-specific code annotations vs the Standard form
- In what cases attributes are handy
- C++11 and C++14 attributes
- New additions in C++17

Why Do We Need Attributes?

Have you ever used `__declspec`, `__attribute__` or `#pragma` directives in your code?

For example:

```
// set an alignment
struct S { short f[3]; } __attribute__ ((aligned (8)));

// this function won't return
void fatal () __attribute__ ((noreturn));
```

Or for DLL import/export in MSVC:

```
#if COMPILING_DLL
    #define DLLEXPORT __declspec(dllexport)
#else
    #define DLLEXPORT __declspec(dllimport)
#endif
```

Those are existing forms of compiler-specific attributes/annotations.

So what is an attribute?

An attribute is additional information that can be used by the compiler to produce code. It might be utilised for optimisation or some specific code generation (like DLL stuff, OpenMP, etc.). Also, annotations allow you to write more expressive syntax and help other developers to reason about code.

Contrary to other languages such as C#, in C++, the compiler has fixed the meta-information system. You cannot add user-defined attributes. In C# you can derive from `System.Attribute`.

What's best about Modern C++ attributes?

Since C++11, we get more and more standardised attributes that will work with other compilers. We're moving away from compiler-specific annotation to standard forms. Rather than learning various annotation syntaxes you'll be able to write code that is common and has the same behaviour.

In the next section, you'll see how attributes used to work before C++11.

Before C++11

In the era of C++98/03, each compiler introduced its own set of annotations, usually with a different keyword.

Often, you could see code with `#pragma`, `__declspec`, `__attribute` spread throughout the code.

Here's the list of the common syntax from GCC/Clang and MSVC:

GCC Specific Attributes

GCC uses annotation in the form of `__attribute__((attr_name))`. For example:

```
int square (int) __attribute__ ((pure)); // pure function
```

Documentation:

- Attribute Syntax - Using the GNU Compiler Collection (GCC)[1]
- Using the GNU Compiler Collection (GCC): Common Function Attributes[2]

MSVC Specific Attributes

Microsoft mostly used `__declspec` keyword, as their syntax for various compiler extensions. See the documentation here: `__declspec` Microsoft Docs[3].

```
__declspec(deprecated) void LegacyCode() { }
```

Clang Specific Attributes

Clang, as it's straightforward to customise, can support different types of annotations, so look at the documentation to find more. Most of GCC attributes work with Clang.

See the documentation here: Attributes in Clang — Clang documentation[4].

[1] https://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/Attribute-Syntax.html
[2] https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#Common-Function-Attributes
[3] https://docs.microsoft.com/en-Us/cpp/cpp/declspec
[4] https://clang.llvm.org/docs/AttributeReference.html

Attributes in C++11 and C++14

C++11 took one big step to minimise the need to use vendor-specific syntax. By introducing the standard format, we can move a lot of compiler-specific attributes into the universal set.

C++11 provides a cleaner format of specifying annotations over our code.

The basic syntax is just `[[attr]]` or `[[namespace::attr]]`.

You can use `[[attr]]` over almost anything: types, functions, enums, etc., etc.

For example:

```
[[attrib_name]] void foo() { }      // on a function
struct [[deprecated]] OldStruct { } // on a struct
```

In C++11 we have the following attributes:

`[[noreturn]]`:

It tells the compiler that control flow will not return to the caller. Examples:

- `[[noreturn]] void terminate() noexcept;`
- functions like `std::abort` or `std::exit` are also marked with this attribute.

`[[carries_dependency]]`:

Indicates that the dependency chain in release-consume `std::memory_order` propagates in and out of the function, which allows the compiler to skip unnecessary memory fence instructions. Mostly to help to optimise multi-threaded code and when using different memory models.

C++14 added:

`[[deprecated]]` and `[[deprecated("reason")]]`:

Code marked with this attribute will be reported by the compiler. You can set its reason.

Example of `[[deprecated]]`:

```
[[deprecated("use AwesomeFunc instead")]] void GoodFunc() { }

// call somewhere:
GoodFunc();
```

GCC reports the following warning:

```
warning: 'void GoodFunc()' is deprecated: use AwesomeFunc instead
[-Wdeprecated-declarations]
```

You know a bit about the old approach, new way in C++11/14... so what's the deal with C++17?

C++17 Additions

With C++17 we get three more standard attributes:

- `[[fallthrough]]`
- `[[nodiscard]]`
- `[[maybe_unused]]`

Extra Info

The new attributes were specified in P0188[5] and P0068[6](reasoning).

Plus three supporting features:

- Attributes for Namespaces and Enumerators
- Ignore Unknown Attributes
- Using Attribute Namespaces Without Repetition

Let's go through the new attributes first.

[5] https://wg21.link/p0188
[6] https://wg21.link/p0068

`[[fallthrough]]` Attribute

Indicates that a fall-through in a switch statement is intentional and a warning should not be issued for it.

```
switch (c) {
case 'a':
    f(); // Warning! fallthrough is perhaps a programmer error
case 'b':
    g();
[[fallthrough]]; // Warning suppressed, fallthrough is ok
case 'c':
    h();
}
```

With this attribute, the compiler can understand the intentions of a programmer. It's also much more readable than using a comment.

`[[maybe_unused]]` Attribute

Suppresses compiler warnings about unused entities:

```
static void impl1() { ... } // Compilers may warn when function not called
[[maybe_unused]] static void impl2() { ... } // Warning suppressed

void foo() {
   int x = 42; // Compilers may warn when x is not used later
   [[maybe_unused]] int y = 42; // Warning suppressed for y
}
```

Such behaviour is helpful when some of the variables and functions are used in debug only path. For example in `assert()` macros;

```
void doSomething(std::string_view a, std::string_view b) {
    assert(a.size() < b.size());
}
```

If later `a` or `b` is no used in this function, then the compiler will generate a warning in release only builds. Marking the given argument with `[[maybe_unused]]` will solve this warning.

`[[nodiscard]]` Attribute

`[[nodiscard]]` can be applied to a function or a type declaration to mark the importance of the returned value:

```
[[nodiscard]] int Compute();
void Test() {
    Compute(); // Warning! return value of a
               // nodiscard function is discarded
}
```

If you forget to assign the result to a variable, then the compiler should emit a warning.

What it means is that you can force users to handle errors. For example, what happens if you forget about using the return value from `new` or `std::async()`?

Additionally, the attribute can be applied to types. One use case for it might be error codes:

```
enum class [[nodiscard]] ErrorCode {
    OK,
    Fatal,
    System,
    FileIssue
};

ErrorCode OpenFile(std::string_view fileName);
ErrorCode SendEmail(std::string_view sendto,
                    std::string_view text);
ErrorCode SystemCall(std::string_view text);
```

Now, every time you'd like to call such functions, you're "forced" to check the return value. For important functions checking return codes might be crucial and using `[[nodiscard]]` might save you from a few bugs.

You might also ask what it means "not to use" a return value?

In the Standard, it's defined as "Discarded-value expressions"[7]. It means that you call a function only for its side effects. In other words, there's no if statement around or an assignment expression. In that case, when a type is marked as `[[nodiscard]]` the compiler is encouraged to report a warning.

However, to suppress the warning you can explicitly cast the return value to `void` or use `[[maybe_unused]]`:

[7] http://en.cppreference.com/w/cpp/language/expressions#Discarded-value_expressions

```
[[nodiscard]] int Compute();
void Test() {
    static_cast<void>(Compute()); // fine...

    [[maybe_unused]] auto ret = Compute();
}
```

In addition, in C++20 the Standard Library will use `[[nodiscard]]` in a few places like: `operator new`, `std::async()`, `std::allocate()`, `std::launder()`, and `std::empty()`.
This feature was already merged into C++20 with P0600[8].

The second addition to C++20 is `[[nodiscard("reason")]]`, see in P1301[9]. This lets you specify why not using a returned value might generate issues — for example, some resource leak.

Attributes for Namespaces and Enumerators

The idea for attributes in C++11 was to be able to apply them to all sensible places like classes, functions, variables, typedefs, templates, enumerations... But there was an issue in the specification that blocked attributes when they were applied on namespaces or enumerators.

This is now fixed in C++17. We can now write:

```
namespace [[deprecated("use BetterUtils")]] GoodUtils {
    void DoStuff() { }
}

namespace BetterUtils {
    void DoStuff() { }
}

// use:
GoodUtils::DoStuff();
```

[8] https://wg21.link/p0600
[9] https://wg21.link/P1301

Clang reports:

```
warning: 'GoodUtils' is deprecated: use BetterUtils
[-Wdeprecated-declarations]
```

Another example is the use of deprecated attribute on enumerators:

```
enum class ColorModes {
    RGB [[deprecated("use RGB8")]],
    RGBA [[deprecated("use RGBA8")]],
    RGB8,
    RGBA8
};

// use:
auto colMode = ColorModes::RGBA;
```

Under GCC we'll get:

```
warning: 'RGBA' is deprecated: use RGBA8
[-Wdeprecated-declarations]
```

Extra Info

The change was described in N4266[10](wording) and N4196[11](reasoning).

Ignore Unknown Attributes

The feature is mostly for clarification.

Before C++17, if you tried to use some compiler-specific attribute, you might even get an error when compiling in another compiler that doesn't support it. Now, the compiler omits the attribute specification and won't report anything (or just a warning). This wasn't mentioned in the Standard, and it needed clarification.

[10]https://wg21.link/n4266
[11]https://wg21.link/n4196

```
// compilers which don't
// support MyCompilerSpecificNamespace will ignore this attribute
[[MyCompilerSpecificNamespace::do_special_thing]]
void foo();
```

For example in GCC 7.1 there's a warnings:

```
warning: 'MyCompilerSpecificNamespace::do_special_thing'
scoped attribute directive ignored [-Wattributes]
void foo();
```

Extra Info

The change was described in P0283R2[12](wording) and P0283R1[13](reasoning).

Using Attribute Namespaces Without Repetition

The feature simplifies the case where you want to use multiple attributes, like:

```
void f() {
    [[rpr::kernel, rpr::target(cpu,gpu)]] // repetition
    doTask();
}
```

Proposed change:

```
void f() {
    [[using rpr: kernel, target(cpu,gpu)]]
    doTask();
}
```

That simplification might help when building tools that automatically translate annotated code of that type into different programming models.

Extra Info

The change was described in: P0028R4[14].

[12] https://wg21.link/p0283r2
[13] https://wg21.link/p0283r1
[14] http://wg21.link/p0028r4

Section Summary

Attributes available in C++17:

Attribute	Description
`[[noreturn]]`	a function does not return to the caller
`[[carries_dependency]]`	extra information about dependency chains
`[[deprecated]]`	an entity is deprecated
`[[deprecated("reason")]]`	provides additional message about the deprecation
`[[fallthrough]]`	indicates a intentional fall-through in a switch statement
`[[nodiscard]]`	a warning is generated if the return value is discarded
`[[maybe_unused]]`	an entity might not be used in the code

Each compiler vendor can specify their syntax for attributes and annotations. In Modern C++, the ISO Committee tries to extract common parts and standardise it as `[[attributes]]`.

There's also a relevant quote from Bjarne Stroustrup's C++11 FAQ[15] about suggested use:

> There is a reasonable fear that attributes will be used to create language dialects. The recommendation is to use attributes to only control things that do not affect the meaning of a program but might help detect errors (e.g. `[[noreturn]]`) or help optimisers (e.g. `[[carries_dependency]]`).

Compiler support

Feature	GCC	Clang	MSVC	C++ Builder
`[[fallthrough]]`	7.0	3.9	VS 2017 15.0	10.3 Rio
`[[nodiscard]]`	7.0	3.9	VS 2017 15.3	10.3 Rio
`[[maybe_unused]]`	7.0	3.9	VS 2017 15.3	10.3 Rio
Attributes for namespaces and enumerators	4.9(namespaces)/6(enums)	3.4	VS 2015 14.0	10.3 Rio
Ignore unknown attributes	yes	3.9	VS 2015 14.0	10.3 Rio
Using attribute namespaces without repetition	7.0	3.9	VS 2017 15.3	10.3 Rio

[15] http://stroustrup.com/C++11FAQ.html#attributes

Appendix A - Compiler Support

If you work with the latest version of a compiler like GCC, Clang or MSVC you may assume that C++17 is fully supported (with some exceptions to the STL implementations). GCC implemented the full support in the version 7.0, Clang did it in the version 6.0 and MSVC is conformant as of VS 2017 15.7 For completeness, here you have a list of features and versions of compilers where it was added.

The up to date resource with the status of the features: CppReference - Compiler Support[16]

GCC

Language[17]

The Library - LibSTDC++[18]

Clang

Language[19]

The Library - LibC++[20]

VisualStudio - MSVC

Announcing: MSVC Conforms to the C++ Standard[21]

C++17/20 Features and Fixes in Visual Studio 2019[22]

STL Features and Fixes in VS 2017 15.8[23]

C++ Builder & RAD Studio

Modern C++ Language Features Compliance Status - RAD Studio[24]

[16] https://en.cppreference.com/w/cpp/compiler_support
[17] https://gcc.gnu.org/projects/cxx-status.html#cxx17
[18] https://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html#status.iso.2017
[19] http://clang.llvm.org/cxx_status.html#cxx17
[20] http://libcxx.llvm.org/cxx1z_status.html
[21] https://blogs.msdn.microsoft.com/vcblog/2018/05/07/announcing-msvc-conforms-to-the-c-standard/
[22] https://devblogs.microsoft.com/cppblog/cpp17-20-features-and-fixes-in-vs-2019/
[23] https://devblogs.microsoft.com/cppblog/stl-features-and-fixes-in-vs-2017-15-8/
[24] http://docwiki.embarcadero.com/RADStudio/Rio/en/Modern_C%2B%2B_Language_Features_Compliance_Status

Compiler Support of C++17 Features

Fixes and Deprecation

Feature	GCC	Clang	MSVC	C++ Builder
Removing `register` keyword	7.0	3.8	VS 2017 15.3	10.3 Rio
Remove Deprecated `operator++`(bool)	7.0	3.8	VS 2017 15.3	10.3 Rio
Removing Deprecated Exception Specifications	7.0	4.0	VS 2017 15.5	10.3 Rio
Removing trigraphs	5.1	3.5	VS 2010	10.3 Rio
New auto rules for direct-list-initialisation	5.0	3.8	VS 2015	10.3 Rio
`static_assert` with no message	6.0	2.5	VS 2017	10.3 Rio
Different begin and end types in range-based for	6.0	3.6	VS 2017	10.3 Rio

Clarification

Feature	GCC	Clang	MSVC	C++ Builder
Stricter expression evaluation order	7.0	4.0	VS 2017	10.3 Rio
Guaranteed copy elision	7.0	4.0	VS 2017 15.6	10.3 Rio
Exception specifications part of the type system	7.0	4.0	VS 2017 15.5	10.3 Rio
Dynamic memory allocation for over-aligned data	7.0	4.0	VS 2017 15.5	10.3 Rio

General Language Features

Feature	GCC	Clang	MSVC	C++ Builder
Structured Binding Declarations	7.0	4.0	VS 2017 15.3	10.3 Rio
Init-statement for if/switch	7.0	3.9	VS 2017 15.3	10.3 Rio
Inline variables	7.0	3.9	VS 2017 15.5	10.3 Rio
`constexpr` Lambda Expressions	7.0	5.0	VS 2017 15.3	10.3 Rio
Lambda Capture of `*this`	7.0	3.9	VS 2017 15.3	10.3 Rio
Nested namespaces	6.0	3.6	VS 2015	10.3 Rio
`has_include`	5	Yes	VS 2017 15.3	10.3 Rio

Templates

Feature	GCC	Clang	MSVC	C++ Builder
Template argument deduction for class templates	7.0/8.0	5.0	VS 2017 15.7	10.3 Rio
Deduction Guides in the Standard Library	8.0	7.0	VS 2017 15.7	10.3 Rio
Declaring non-type template parameters with auto	7.0	4.0	VS 2017 15.7	10.3 Rio
Fold expressions	6.0	3.9	VS 2017 15.5	10.3 Rio
`if constexpr`	7.0	3.9	VS 2017	10.3 Rio

Attributes

Feature	GCC	Clang	MSVC	C++ Builder
`[[fallthrough]]`	7.0	3.9	VS 2017 15.0	10.3 Rio
`[[nodiscard]]`	7.0	3.9	VS 2017 15.3	10.3 Rio
`[[maybe_unused]]`	7.0	3.9	VS 2017 15.3	10.3 Rio
Attributes for namespaces and enumerators	4.9(namespaces)/6(enums)	3.4	VS 2015 14.0	10.3 Rio
Ignore unknown attributes	yes	3.9	VS 2015 14.0	10.3 Rio
Using attribute namespaces without repetition	7.0	3.9	VS 2017 15.3	10.3 Rio

The Standard Library

Feature	GCC	Clang	MSVC	C++ Builder
`std::optional`	7.1	4.0	VS 2017 15.0	10.3 Rio
`std::variant`	7.1	4.0	VS 2017 15.0	10.3 Rio
`std::any`	7.1	4.0	VS 2017 15.0	10.3 Rio
`std::string_view`	7.1	4.0	VS 2017 15.0	10.3 Rio
String Searchers	7.1	5.0	VS 2017 15.3	no
String Conversions	8 (only integral types)	in progress	VS 2017 15.8	no
Parallel Algorithms	9.1	in progress	VS 2017 15.7	no
Filesystem	8.0	7.0	VS 2017 15.7	10.3 Rio

Other STL changes

Feature	GCC	Clang	MSVC	C++ Builder
`std::byte`	7.1	5.0	VS 2017 15.3	10.3 Rio
Improvements for Maps and Sets	7.0	3.9	VS 2017 15.5	no
`insert_or_assign()`/`try_emplace()` for maps	6.1	3.7	VS 2017 15	no
Emplace Return Type	7.1	4.0	VS 2017 15.3	no
Sampling algorithms	7.1	In Progress	VS 2017 15	no
`gcd` and `lcm`	7.1	4.0	VS 2017 15.3	10.3 Rio
`clamp`	7.1	3.9	VS 2015.3	10.3 Rio
Special Mathematical Functions	7.1	Not yet	VS 2017 15.7	10.3 Rio
Shared Pointers and Arrays	7.1	In Progress	VS 2017 15.5	no
Non-member `size()`, `data()` and `empty()`	6.1	3.6	VS 2015	no
`constexpr` Additions to the Standard Library	7.1	4.0	VS 2017 15.3	no
`scoped_lock`	7.1	5.0	VS 2017 15.3	no
Polymorphic Allocator & Memory Resource	9.1	In Progress	VS 2017 15.6	no

Appendix B - Resources and References

You can purchase the official C++17 Standard at the ISO site:

ISO/IEC 14882:2017 - Programming languages - C++[25]

However, you can also read the free draft that's very close to the published version:

N4687, 2017-07-30, **Working Draft, Standard for Programming Language C++**[26]

This PDF is from isocpp.org[27]. Also, have a look here for more information about the papers and the status of C++: isocpp/Standard C++[28].

For a quick overview of C++17 changes, here's a handy list located at:

P0636 - Changes between C++14 and C++17[29]

Books:

- **C++17 - The Complete Guide** by Nicolai Josuttis
- **C++17 STL Cookbook** by Jacek Galowicz
- **Modern C++ Programming Cookbook** by Marius Bancila
- **C++ Templates: The Complete Guide** (2nd Edition) by David Vandevoorde, Nicolai M. Josuttis, Douglas Gregor
- **Professional C++, 4th Edition** by Marc Gregoire

General C++ Links:

- Compiler support: C++ compiler support
- ISO Standard C++

[25] https://www.iso.org/standard/68564.html
[26] https://wg21.link/n4687
[27] https://isocpp.org/
[28] https://isocpp.org/std/the-standard
[29] https://wg21.link/P0636

- Jason Turner: C++ Weekly channel, where he covered most (or even all!) of C++17 features.
- Simon Brand blog - with lot's of information about C++17
- Arne Mertz blog
- Rainer Grimm Blog
- CppCast
- FluentC++

General C++17 Language Features

- Simon Brand: Template argument deduction for class template constructors
- Class template deduction(since C++17) - cppreference.
- "Using fold expressions to simplify variadic function templates" in Modern C++ Programming Cookbook.
- Simon Brand: Exploding tuples with fold expressions
- Baptiste Wicht: C++17 Fold Expressions
- Fold Expressions - ModernesCpp.com
- Adding C++17 structured bindings support to your classes
- C++ Weekly Special Edition - Using C++17's constexpr if - YouTube - real examples from Jason and his projects.
- C++17: let's have a look at the constexpr if – FJ
- C++ 17 vs. C++ 14 — if-constexpr – LoopPerfect – Medium
- Two-phase name lookup support comes to MSVC
- What does the `carries_dependency` attribute mean? - Stack Overflow
- Value Categories in C++17 – Barry Revzin – Medium
- Rvalues redefined | Andrzej's C++ blog
- Guaranteed Copy Elision Does Not Elide Copies - MSVC C++ Team Blog

Expression Evaluation Order:

- GotW #56: Exception-Safe Function Calls
- Core Guidelines: ES.43: Avoid expressions with undefined order of evaluation
- Core Guidelines: ES.44: Don't depend on order of evaluation of function arguments

About `std::optional`:

- Andrzej's C++ blog: Efficient optional values
- Andrzej's C++ blog: Ref-qualifiers
- Clearer interfaces with `optional<T>` - Fluent C++
- Optional - Performance considerations - Boost 1.67.0
- Enhanced Support for Value Semantics in C++17 - Michael Park, CppCon 2017
- std::optional: How, when, and why | Visual C++ Team Blog

About `std::variant`:

- SimplifyC++ - Overload: Build a Variant Visitor on the Fly.
- Variant Visitation by Michael Park
- Sum types and state machines in C++17
- Implementing State Machines with std::variant
- Pattern matching in C++17 with std::variant, std::monostate and std::visit
- Another polymorphism | Andrzej's C++ blog
- Inheritance vs std::variant, C++ Truths

About `string_view`

- CppCon 2015 `string_view` — Marshall Clow
- string_view odi et amo - Marco Arena
- C++17 string_view – Steve Lorimer
- Modernescpp - string_view
- J. Müller - std::string_view accepting temporaries: good idea or horrible pitfall?
- abseil / Tip of the Week #1: string_view
- `std::string_view` is a borrow type – Arthur O'Dwyer – Stuff mostly about C++
- C++ Russia 2018: Victor Ciura, Enough string_view to hang ourselves - YouTube
- StringViews, StringViews everywhere! - Marc Mutz - Meeting C++ 2017 - YouTube
- Jacek's C++ Blog · Const References to Temporary Objects
- abseil / Tip of the Week #107: Reference Lifetime Extension
- C++17 - Avoid Copying with std::string_view - ModernesCpp.com

String Conversions and Searchers

- How to Convert a String to an int in C++ - Fluent C++
- How to *Efficiently* Convert a String to an int in C++ - Fluent C++
- How to encode char in 2-bits? - Stack Overflow

About Filesystem

- Chapter 7, "Working with Files and Streams" - of **Modern C++ Programming Cookbook**.
- examples like: Working with filesystem paths, Creating, copying, and deleting files and directories, Removing content from a file, Checking the properties of an existing file or directory, searching.
- Chapter 10 ""Filesystem" from "**C++17 STL Cookbook**"
- examples: path normalizer, Implementing a grep-like text search tool, Implementing an automatic file renamer, Implementing a disk usage counter, statistics about file types, Implementing a tool that reduces folder size by substituting duplicates with symlinks
- C++17- std::byte and std::filesystem - ModernesCpp.com
- How similar are Boost filesystem and the standard C++ filesystem libraries? - SO
- bfilipek.com: Converting from Boost to std::filesystem

Parallel Algorithms

- Bryce Adelstein's talk about parallel algorithms. Contains a lot of examples for map reduce (transform reduce) algorithm: CppCon 2016: Bryce Adelstein Lelbach "The C++17 Parallel Algorithms Library and Beyond" - YouTube
- Sean Parent – Better Code: Concurrency - code::dive 2016
- Simon Brand - std::accumulate vs. std::reduce
- Using C++17 Parallel Algorithms for Better Performance | Visual C++ Team Blog
- bfilipek.com: The Amazing Performance of C++17 Parallel Algorithms, is it Possible?

The Full Version of The Book

The text you read contains only extracted parts of the final book. However, you can purchase the full version in three ways: as a paperback, as an ebook, or as an interactive course.

Amazon (Paperback)

Amazon.us, Amazon.co.uk, Amazon.de, Amazon.fr, Amazon.es, Amazon.jp, Amazon.ca

Leanpub (ebook) with 25% off discount!

C++17 In Detail @Leanpub - with 25% off discount!

Educative Interactive Course

C++17 in Detail: A Deep Dive @Educative

It consists of more than 220 lessons, many quizzes, code snippets and what's best is that it has 123 playgrounds! That means you can compile and edit code samples directly in the browser, so there's no need for you to switch back and forth to some compiler/IDE.

THIS EBOOK

C++17 IN DETAIL

PART I - LANGUAGE FEATURES

IS OFFERED TO YOU BY EMBARCADERO AND WHOLE TOMATO SOFTWARE

Embarcadero is the creator of the C++Builder, the powerful C++ IDE for multi-platform development focusing on database access and excellent UI design for all your platforms.

Find out more about C++Builder here.

WholeTomato is the creator of Visual Assist, an extension that improves Visual C# and Visual C++with a multitude of productivity features above those provided in the default IDE. It's especially useful for very large codebases!

Find out more about Visual Assist here.

Happy programming from all of us at Embarcadero and Whole Tomato!